AUTOMATIC MODULAR ABSTRACTIONS FOR TEMPLATE NUMERICAL CONSTRAINTS

DAVID MONNIAUX

CNRS / VERIMAG, Centre Équation, 2, avenue de Vignate, 38610 Gières cedex, France
e-mail address: David.Monniaux@imag.fr

Abstract. We propose a method for automatically generating abstract transformers for static analysis by abstract interpretation. The method focuses on linear constraints on programs operating on rational, real or floating-point variables and containing linear assignments and tests. Given the specification of an abstract domain, and a program block, our method automatically outputs an implementation of the corresponding abstract transformer. It is thus a form of program transformation.

In addition to loop-free code, the same method also applies for obtaining least fixed points as functions of the precondition, which permits the analysis of loops and recursive functions.

The motivation of our work is data-flow synchronous programming languages, used for building control-command embedded systems, but it also applies to imperative and functional programming.

Our algorithms are based on quantifier elimination and symbolic manipulation techniques over linear arithmetic formulas. We also give less general results for nonlinear constraints and nonlinear program constructs.

1. Introduction

Program analysis consists in deriving properties of the possible executions of a program from an algorithmic processing of its source or object code. Example of interesting properties include: "the program always terminates"; "the program never executes a division by zero"; "the program always outputs a well-formed XML document"; "variable x always lies between 1 and 3". There has been a considerable amount of work done since the late 1970s on *sound* methods of program analysis — that is, methods that produce results that are guaranteed to hold for all program executions, as opposed to *bug finding* methods such as program testing, which cannot provide such guarantees in general.

Static analysis by *abstract interpretation* is one of the various approaches to sound program analysis. Grossly speaking, abstract interpretation casts the problem of obtaining

Received by the editors May 26, 2010.

VERIMAG is a joint research laboratory of CNRS, Université Joseph Fourier and Grenoble-INP.

This work was partially funded by the "ASOPT" project of the *Agence nationale de la recherche* (ANR).

supersets of the set of reachable states of programs into a problem of finding fixed points of certain monotone operators on certain ordered sets, known as *abstract domains*. When dealing with programs operating on arithmetic values (integer or real numbers, or, more realistically, bounded integers and floating-point values), these sets are often defined by numerical constraints, and ordered by inclusion. One may, for instance, attempt to compute, for each program point and each variable, an interval that is guaranteed to contain all possible values of that variable at that point. The problem is of course how to compute these fixed points. Obviously, the smaller the intervals, the better, so we would like to compute them as small as possible. Ideally, we would like to compute the least fixed point, that is, the least inductive invariant that can be expressed using intervals.

The purpose of this article is to expose how to compute such least fixed points exactly, at least for certain classes of programs and certain abstract domains. Specifically, we consider programs operating over real variables using only linear comparisons (e.g. $x + 2y \leq 3$ but not $x^2 \leq y$), and abstract domains defined using a finite number of linear constraints $\sum_i a_i v_i \leq C$, where the a_i are fixed coefficients, C is a parameter (whose computation is the goal of the analysis) and the v_i are the program variables. Such domains evidently include the intervals, where the constraints are of the form $v_i \leq C$ and $-v_i \leq C$.

Not only can we compute such least fixed points exactly if all parameters are known, but we can also deal with the case where some of the parameters are unknowns, in which case we obtain the parameters of the least fixed point as explicit, algorithmic, functions of the unknowns. We can thus generate, once and for all, the *abstract transformers* for blocks of code: that is, those that map the parameters in the precondition of the block of code to a suitable postcondition. For instance, in the case of interval analysis, we derive explicit functions mapping the bounds on variables at the entrance of a loop-free block to the tightest bounds at the outcome of the block, or to the least inductive interval invariant of a loop. This allows *modular analysis*: it is possible to analyse functions or other kind of program modules (such as nodes in synchronous programming) in isolation of the rest of the code.

A crucial difference with other methods, even on loop-free code, is that we derive the optimal — that is, the most precise — abstract transformer for the whole sequence. In contrast, most static analyses only have optimal transformers for individual instructions; they build transformers for whole sequences of code by composition of the transformers for the individual instructions. Even on very simple examples, the optimal transformer for a sequence is not the composition of the optimal transformers of the individual instructions. Furthermore, most other methods are forced to merge the information from different execution traces at the end of "if-then-else" and other control flow constructs in a way that loses precision. In contrast, our method distinguishes different paths in the control flow graph as long as they do not go through loop iteration.

Our approach is based on *quantifier elimination*, a technique operating on logical formulas that transforms a formula with quantifiers into an equivalent quantifier-free formula: for instance, $\forall x\ (x \geq y \Rightarrow x \geq 1)$ is turned into the equivalent $y \geq 1$. Essentially, we define the result that we would like to obtain using a quantified logical formula, and, using quantifier elimination and further formula manipulations, we obtain an algorithm that computes this result. Thus, one can see our method as automatically transforming a non executable *specification* of the result of the analysis into an algorithm computing that result.

In §2, we shall provide background about abstract interpretation and quantifier elimination. In §3, we shall provide the main results of the article, that is, how to derive optimal abstract transformers for template linear constraint domains. In §4, we shall investigate various extensions to that framework, still based on quantifier elimination in linear real arithmetic; e.g. how to deal with floating-point values. In §5, we shall explain how to move from the single loop case to more complex control flow. In §6, we shall describe our implementation of the main algorithm and some limited extensions. In §7, we shall investigate extensions of the same framework using quantifier elimination on other theories, namely Presburger arithmetic and the theory of real closed fields (nonlinear real arithmetic). In §8, we shall list some related work and compare our method to other relevant approaches. Finally, §9 concludes.

This article is based upon two conference articles [82, 83].

2. Background

In this section, we shall recall a few definitions, notations and results on program analysis by abstract interpretation, then quantifier elimination.

2.1. Abstract interpretation.

It is well-known that, in the general case, fully automatic program analysis is impossible for any nontrivial property.[1] Thus, all analysis methods must have at least one of the following characteristics:

- They may bound the memory size of the studied program, which then becomes a finite automaton, on which most properties are decidable. *Explicit-state model-checking* works by enumerating all reachable states, which is tractable only while *implicit state model-checking* represents sets of states using data structures such as binary decision diagrams [27].
- They may restrict the programming language used, making it not Turing-complete, so that properties become decidable. For instance, reachability in pushdown automata is decidable even though their memory size is unbounded [17].
- They may restrict the class of properties expressed to properties of bounded executions; e.g., "within the first 10000 steps of execution, there is no division by zero", as in *bounded model checking* [13].
- They may be *unsound* as proof methods: they may fail to detect that the desired property is violated. Typically, *bug-finding* and testing programs are in that category, because they may fail to detect a bug. Some such analysis techniques are not based on program semantics, but rather on finding patterns in the program syntax [38].
- They may be *incomplete* as proof methods: they may fail to prove that a certain property holds, and report spurious violations. Methods based on abstraction fall in that category.

Abstraction by over-approximation consists in replacing the original problem, undecidable or very difficult to decide, by a simpler "abstract" problem whose behaviours are guaranteed to include all behaviours of the original problem.

[1]This result, formally given within the framework of recursive function theory, is known as Rice's theorem [96, p. 34][92, corollary B]. It is obtained by generalisation from Turing's halting theorem. Interpreted upon program semantics, the theorem states that the only properties of the denotational semantics of programs that can be algorithmically decided on the source code are the trivial properties: uniformly "true" or uniformly "false".

Listing 1: Original program

```
int x, y;
bool a;
if (x < y) a = false;
```

Listing 2: Boolean program

```
bool a;
if (nondet()) a = false;
```

Figure 1: Transformation of a program into a Boolean program by erasing the numeric part and replacing tests over numerical quantities by nondeterministic choice (nondet() nondeterministically returns true or false).

An example of an abstraction is to erase from the program all constructs dealing with numerical and pointer types (or replacing them with nondeterministic choices, if their value is used within a test), keeping only Boolean types (Fig. 1). Obviously, the behaviours of the resulting program encompass all the behaviours of the original program, plus maybe some extra ones.

Further abstraction can be applied to this Boolean program: for instance, the "3-value logic" abstraction [91] which maps any input or output variable to an abstract parameter taking its value in a 3-element set: "is 0", "is 1", "can be 0 or 1";[2] for practical purposes it may be easier to encode these values using a couple of Booleans, respectively meaning "can be 0" and "can be 1", thus the abstract values $(1, 0)$, $(0, 1)$ and $(1, 1)$. The abstract value $(0, 0)$ obtained for any variable at a program point means that this program point is unreachable. Given a vector of input abstract parameters, one for each input variable of the program, the *forward abstract transfer function* gives a correct vector of output abstract parameters, one for each output variable of the program. Quite obviously, in the absence of loops, it is possible to obtain a suitable forward abstract transfer function by applying simple logical rules to the Boolean function defined by the Boolean program. An effective implementation of the forward abstract transfer function can thus be obtained by a transformation of the source program.

For programs operating over numerical quantities, a common abstraction is *intervals.* [33, 34] To each input x, one associates an interval $[x_{\min}, x_{\max}]$, to each output x' an interval $[x'_{\min}, x'_{\max}]$. How can one compute the $(x'_{\min}, x'_{\max})_{x \in V}$ bounds from the $(x_{\min}, x_{\max})_{x \in V}$? The most common method is *interval arithmetic*: to each elementary arithmetic operation, one attaches an abstract counterpart that gives bounds on the output of the operation given bounds on the inputs. For instance, if one knows $[a_{\min}, a_{\max}]$ and $[b_{\min}, b_{\max}]$, and $c = a + b$, then one computes $[c_{\min}, c_{\max}]$ as follows: $c_{\min} = a_{\min} + b_{\min}$ and $c_{\max} = a_{\max} + b_{\max}$. If a program point can be reached by several paths (e.g. at the end of an if-then-else construct), then a suitable interval $[x_{\min}, x_{\max}]$ can be obtained by a *join* of all the intervals for x at the end of these paths: $[a, b] \sqcap [c, d] = [\min(a, c), \max(b, d)]$. Again, for a loop-free program, one can obtain a suitable effective forward abstract transfer function by a program transformation of the source code.

The abstract transfer function defined by interval arithmetic is always correct, but is not necessarily the most precise. For instance, on example in Fig. 2, the best abstract transfer function maps any input range to $z_{\min} = z_{\max} = 0$, since the output z is always zero,

[2]For brevity, we identify "false" with 0 and "true" with 1.

Listing 3: Program computing zero

```
double x, y, z;
/* x lies in [x_min, x_max]} */
y = x;
z = x−y;
```

Listing 4: Interval transformer

```
y_min = x_min;
y_max = x_max;
z_min = x_min − y_max;
z_max = x_max − y_min;
```

Figure 2: The transformer for the interval domain obtained by composition of locally optimal abstract transformers is imprecise. For each statement (on the left) we use a corresponding optimal transformer (on the right), but the composition of these transformers is not optimal. For the sake of simplicity, all variables are considered to be real numbers.

while the one obtained by applying interval arithmetic to all program statements yields, in general, larger intervals. The weakness of the interval domain on this example is evidently due to the fact that it does not keep track of relationships between variables (here, that $x = y$). *Relational abstract domains* such as the *octagons* [74, 76, 77] or *convex polyhedra* [4, 36, 58] address this issue. Yet, neither octagons nor polyhedra provide analyses that are guaranteed to give optimal results.

Consider the following program:

Listing 5: Undistinguished paths

```
int x;
if (x > 0) x= 1; else x= −1;
if (x == 0) x= 2;
```

Obviously, the second test can never be taken, since x can only be ± 1; however an interval, octagon or polyhedral analysis will conclude, after joining the informations for both branches of the first test, that x lies in $[-1, 1]$ and will not be able to exclude the case $x = 0$. The final invariant will therefore be $x \in [-1, 2]$.

In contrast, our method, applied to this example, will correctly conclude that the optimal output invariant for x is $[-1, 1]$. In fact, our method yields the same result as considering the (potentially exponentially large) set of paths between the beginning and the end of the program, and for each path, computing the least output interval, then computing the join of all these intervals.

We have so far left out programs containing loops; when programs contain loops or recursive functions, a central problem of program analysis is to find *inductive invariants*. In the case of Boolean programs, given constraints on the input parameters, the set of reachable states can be computed exactly by model-checking algorithms; yet, these algorithms do not give a closed-form representation of the abstract transfer function mapping input parameters to output parameters for the 3-value abstraction. In the case of numerical abstractions such as the intervals, octagons or polyhedra, the most common way to find invariants is through the use of a *widening operator* [34, 35].

Intuitively, widening operators observe the sets of reachable states after N and $N + 1$ loop iterations and extrapolate them to a "candidate invariant". For instance, the widening

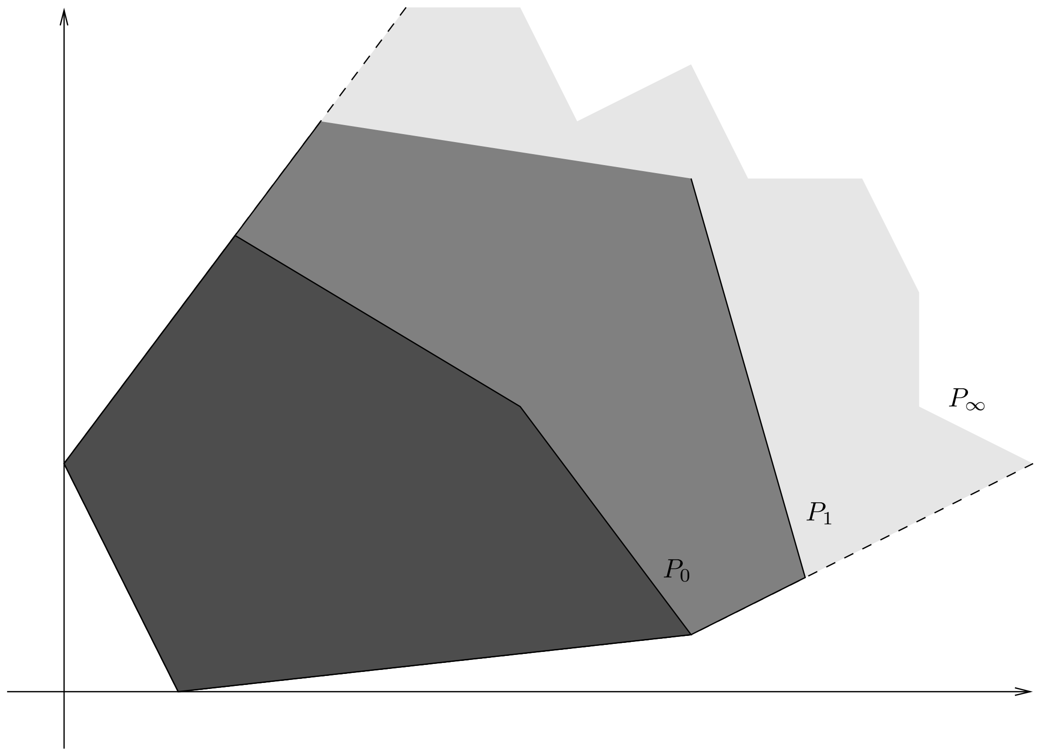

Figure 3: The standard widening on convex polyhedra [36, 58], here demonstrated on polyhedra in dimension 2 (polygons). The widening operator observes the sets of reachable states P_0 and P_1 at two consecutive iterations, and keeps only the constraints (polyhedral faces, here polygon edges) that are stable across iterations. The resulting P_∞ polyhedron is then proposed as an invariant.

operator, observing a sequence of intervals $[0, 1]$, $[0, 2]$, $[0, 3]$ may wish to try $[0, +\infty)$. See Fig. 3 for an example with the standard widening operator on convex polyhedra.

Let u_0 be the set of initial states of a loop, and let $\rightarrow_\tau$ be transition relation for this loop ($\sigma \rightarrow_\tau \sigma'$ means that σ' is reachable in one loop step from σ). The set of reachable states at the head of the loop is the least fixed point of $f : u \mapsto u \cup \{\sigma' \mid \exists \sigma \in u \wedge \sigma \rightarrow_\tau \sigma'\}$, which is obtained as the limit of the ascending sequence defined by $u_{n+1} = f(u_n)$. By abstract interpretation, we replace this sequence by an abstract sequence $u_n^\sharp$ defined by $u_{n+1}^\sharp = f^\sharp(u_n^\sharp)$, such that for any n, $u_n^\sharp$ is an abstraction of u_n. If this sequence is stationary, that is, $u_{N+1}^\sharp = u_N^\sharp$ for some N, then $u_N^\sharp$ is an abstraction of the least fixed point of f and thus of the least invariant of the transition relation τ containing u_0.

When one uses a widening operator $\triangledown$, the function $f^\sharp(u_n^\sharp)$ is defined as $u_n^\sharp \triangledown f_p^\sharp(u_n^\sharp)$ where $f_p^\sharp$ is an abstraction of f. The design of the widening operator ensures convergence of $u_n^\sharp$ in finite time. The exceptions to the use of widening operators are static analysis domains that satisfy the *ascending condition*, such as the domain of linear equality constraints [65] and that of linear congruence constraints [55]: with $f^\sharp = f_p^\sharp$ the sequence $u_n^\sharp$ always becomes stable within finite time.

Listing 6: Circular buffer

```
i = 0;
while (true) {
  if (nondet()) {
    i = i+1;
    if (i >= 10) i=0;
  }
}
```

Again, widening operators provide correct results, but these results can be grossly over-approximated. Much of the literature on applied analysis by abstract interpretation discusses workarounds that give precise results in certain cases: *narrowing* iterations [33, 34], widening "up to" [57, §3.2], "delayed" or with "thresholds" [15], etc.

A simple example of a program with one single variable where narrowing iterations fail to improve precision is Listing 6. This program is a much simplified version of a piece of code maintaining a circular buffer inside a large reactive control loop, where some piece of data is inserted only at certain loop iterations. We only kept the instructions relevant to the array index i and abstracted away the choice of the loop iterations where data is inserted as nondeterministic nondet().

If we analyse this loop using intervals with the standard widening with a widening point at the head of the loop, we obtain the sequence $[0, 0]$, $[0, 1]$, $[0, 2]$ and then widening to $[0, +\infty)$. Narrowing iterations then fail to increase the precision. The reason is that the transition relation for this loop includes the identity function (when nondet() is false), thus the concrete function whose least fixed point defines the set of reachable states [35] satisfies $X \subseteq \phi(X)$ for all X (in other word, ϕ is *expansive*). Thus, once the widening operator overshoots the least fixed point, it can never recover it.

A similar problem is posed by:

```
i = 0;
while (true) {
  i = i+1;
  if (i == 10) i=0;
}
```

The usual widening iterations overshoot to $[0, +\infty)$, and narrowing does not recover from there. Furthermore, this example illustrates how widening makes analysis *non monotonic*: contrary to one could expect, having extra precision on the precondition of a program can result in worse precision for the inferred invariants. For instance, consider the above problem and replace the first line by **assume**(i>=0 && i<=9). Clearly, the resulting program is an abstraction of the above example, since it has strictly more behaviours (we allow $1, \dots, 9$ as initial values for i). Yet, the analysis of the loop will yield a more precise behaviour: the interval $[0, 9]$ is stable and the analysis terminates immediately.

Both these very simple examples can be precisely analysed using *widening up to*[57, §3.2], also known as *widening with thresholds* [15, Sec. 7.1.2]: a preliminary phase collects all constants to which i is compared, and instead of widening to $+\infty$, one widens to the next larger such constant if it exists — in this case, since x < 10 stands for $x \leq 9$, widening with threshold would widen to 9, which is the correct value. This approach is not general — it fails if instead of the constant 10 we have some computed value. Of course, improvements are possible: for instance, one could analyse all the program up to this loop in order to prove that certain variables are constant, then use this information for setting thresholds for further loops. Yet, again, this is not a general approach.

This is the second problem that this article addresses: how to obtain, in general, optimal invariants for certain classes of programs and numerical constraints. Furthermore, our methods provide these invariants as functions of the parameters of the precondition of the loop; this is one difference with our proposal, which computes the best, thus, again, they provide effective, optimal abstract transfer functions for loop constructs.

2.2. **Quantifier elimination.** Consider a set A of atomic formulas. The set $U(A)$ of *quantifier-free formulas* is the set of formulas constructed from A using operators $\wedge$, $\vee$ and $\neg$; the set $Q(A)$ of *quantified formulas* is the set of formulas constructed from A using the above operators and the $\exists$ and $\forall$ quantifiers. Such formulas are thus trees whose leaves are the atomic formulas. A *literal* is an atomic formula or the negation thereof. The set of *free variables* $FV(F)$ of a formula F is defined as usual. A quantifier-free formula without variables is said to be *ground*. A formula without free variables is said to be *closed*; the *existential closure* of a formula F is F with existential quantifiers for all free variables prepended; the *universal closure* is the same with universal quantifiers. A quantifier-free formula is said to be in *disjunctive normal form* (DNF) if it is a disjunction of conjunctions, that is, is of the form $(l_{1,1} \wedge \cdots \wedge l_{1,n_1}) \vee \cdots \vee (l_{m,1} \wedge \cdots \wedge l_{m,n_m})$, and is said to be in *conjunctive normal form* (CNF) if it is a conjunction of disjunctions. Any quantifier-free formula can be converted into CNF or DNF by application of the distributivity laws $(A \vee B) \wedge C \equiv (A \wedge C) \vee (B \wedge C)$ and $(A \wedge B) \vee C \equiv (A \vee C) \wedge (B \vee C)$, though better algorithms exist, such as ALL-SAT modulo theory [81].

2.2.1. *Linear real inequalities.* Let A be the set of linear inequalities with integer or rational coefficients over a set of variables V. By elementary calculus, such inequalities can be equivalently written in the following forms: $l(v_1, v_2, \dots) \geq C$ or $l(v_1, v_2, \dots) > C$, with l a linear expression with integer coefficients over V and C a constant. Let us first consider the theory of *linear real arithmetic* (LRA): models of a formula F are mappings from F to the real field $\mathbb{R}$, and notions of equivalence and satisfiability follow. Note that satisfiability and equivalence are not affected by taking models to be mappings from F to the rational field $\mathbb{Q}$. Deciding whether a LRA formula is satisfiable is, again, a NP-complete problem known as *satisfiability modulo theory* (SMT) of real linear arithmetic. Again, practical implementations, known as SMT-solvers , are capable of dealing with rather large formulas; examples include Yices [45] and Z3 [41].[3]

The theory of linear real arithmetic admits quantifier elimination. For instance, the quantified formula $\forall x\ (x \geq y \implies x \geq 3)$ is equivalent to the quantifier-free formula $y \geq 3$. Quantified linear real arithmetic formulas are thus decidable: by quantifier elimination, one can convert the existential closure of the formula to an equivalent ground formula, the truth of which is trivially decidable by evaluation. The decision problem for quantified formulas

[3]The yearly SMT-COMP competition has SMT-solvers compete on a large set of benchmarks. The SMT-LIB site [9] documents various theories amenable to SMT, including large libraries of benchmarks. Kroening and Strichman [66] is an excellent introductory material to the algorithms behind SMT-solving.

over rational linear inequalities requires at least exponential time, thus quantifier elimination is at least exponential. [19, §7.4]. [48, Th. 3] Weispfenning [107] discusses complexity issues in more detail.

Again, most quantifier elimination algorithms proceed by induction over the structure of the formula, and thus begin by eliminating the innermost quantifiers, progressively replacing branches of the formula containing quantifiers by quantifier-free equivalent branches. By application of the equivalence $\forall x\ F \equiv \neg \exists x\ \neg F$, one can reduce the problem to eliminating existential quantifiers only. Consider now the problem of eliminating the existential quantifiers from $\exists\ x_1 \dots x_n\ F$ where F is quantifier-free. We can first convert into DNF: $\exists x_1 \dots x_n\ (C_1 \vee \dots \vee C_m)$ where the C_i are conjunctions, then to the equivalent formula $(\exists\ x_1 \dots x_n\ C_1) \vee \dots \vee (\exists\ x_1 \dots x_n\ C_m)$. We thus have reduced the quantifier elimination problem for general formula to the problem of quantifier elimination for conjunctions of linear inequalities. Remark that, geometrically, this quantifier elimination amounts to computing the projection of a convex polyhedron along the dimensions associated with the variables $x_1, \dots, x_n$, with the original polyhedron and its projection being defined by systems of linear inequalities. The Fourier-Motzkin elimination procedure [66, §5.4] converts $\exists x_1 \dots x_n\ C$ into an equivalent conjunction. This is what we refer to the "conversion to DNF followed by projection approach". This approach is good for quickly proving that the theory admits quantifier elimination, but it is very inefficient. We shall now see better methods.

Ferrante and Rackoff [47] proposed a substitution method [19, §7.3.1][87, §4.2][107]: a formula of the form $\exists x\ F$ where F is quantifier-free is replaced by an equivalent disjunction $F[e_1/x] \vee \dots \vee F[e_n/x]$, where the e_i are affine linear expressions built from the free variables of $\exists x\ F$. Note the similarity to the naive elimination procedure we described for Boolean variables: even though the existential quantifier ranges over an infinite set of values, it is in fact only necessary to test the formula F at a finite number of points (see Fig. 4). The drawback of this algorithm is that n is proportional to the square of the number of occurrences of x in the formula; thus, the size of the formula can be cubed for each quantifier eliminated. Loos and Weispfenning [69] proposed a *virtual substitution* algorithm[4] [87, §4.4] that works along the same general ideas but for which n is proportional to the number of occurrences of x in the formula. Our benchmarks show that Loos and Weispfenning's algorithm is generally much more efficient than Ferrante and Rackoff's, despite the latter method being better known. [80, 81]

The main drawback of substitution algorithms is that the size of the formulas generally grows very fast as successive variables are eliminated. There are few opportunities for simplifications, save for replacing inequalities equivalent to false (e.g. $0 < 0$) by false and similarly for true, then applying trivial rewrites (e.g. $\text{false} \vee x \rightsquigarrow x$, $\text{false} \wedge x \rightsquigarrow \text{false}$). Our experience is that these algorithms tend to terminate with out-of-memory [80, 81]. Scholl et al. [100] have proposed using and-inverter graphs (AIGs) and SAT modulo theory (SMT) solving in order to simplify formulas and keeping their size manageable.

Another class of algorithms improve on the conversion to DNF then projection approach, by combining both phases: we proposed an eager algorithm based on this idea [81],

[4]This method replaces x by a formula that does not evaluate to a rational number, but to a sum of a rational number and optionally an infinitesimal ε, taken to be a number greater than 0 but less than any positive real; the infinitesimals are then erased by application of the rules governing comparisons. In practical implementations, one does both substitution and erasure of infinitesimals in one single pass, and infinitesimals never actually appear in formulas; thus the phrase *virtual substitution*.

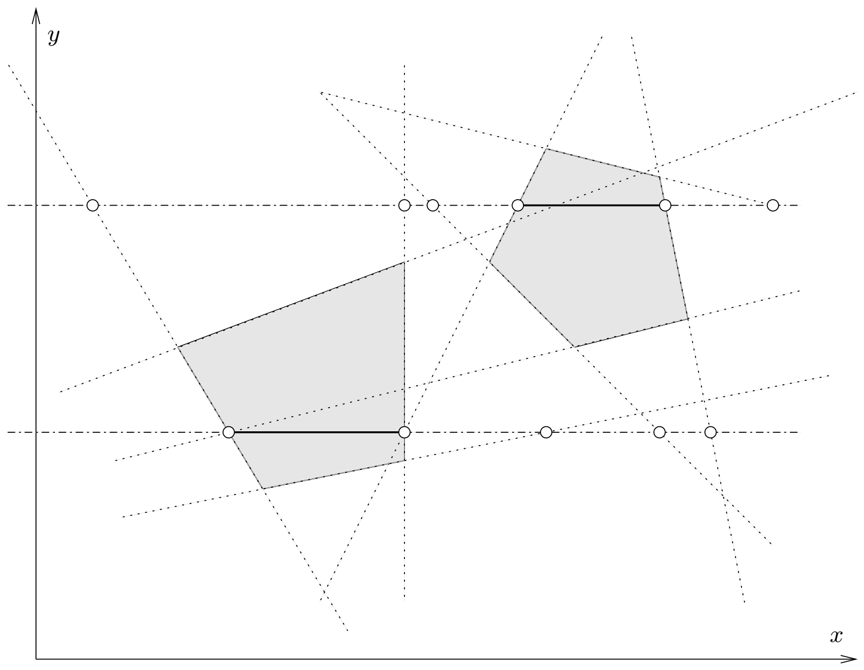

Figure 4: The gray zone S is the set of (x, y) solutions of formula F, whose atoms are the linear inequalities corresponding to the lines Δ drawn with dashes. For fixed $y = y_0$, the set of x such that $F(x, y)$ is true is made of intervals whose ends lie within the set I of intersections of the $y = y_0$ line with the lines in Δ, drawn with a small circle. $y = y_0$ therefore has an intersection with S if and only if a point in I, or an interval with both ends in $I \cup \{-\infty, +\infty\}$, lies within S. This condition can be tested using $x \to \pm\infty$ and all midpoints to intervals with both ends in I, as per Ferrante and Rackoff [47], or, in addition to $I \cup \{-\infty\}$, for any element of I a point infinitesimally close to the right of it, as per Loos and Weispfenning [69]. Both methods exploit the fact that the coordinates of all points from I (intersection of $y = y_0$ and a line from Δ) can be expressed as affine linear functions of y_0.

then a lazy version, which computes parts of formulas as needed [80]. Instead of syntactic conversion to DNF, we use a SMT solver to point to successive elements of the DNF, and instead of using Fourier-Motzkin elimination, which tends to needlessly blow up the size of the formulas, we use libraries for computations over convex polyhedra, which can compute a minimal constraint representation of the projection of a polyhedron given by constraints (that is, inequalities) — which is the geometrical counterpart of performing elimination of a block of existentially quantified variables. Experiments have shown that such methods are generally competitive with substitution approaches, with different classes of benchmarks showing a preference for one of the two families of methods [80]; for the kinds of problems we consider in this article, it seems that the SMT+projection methods are more efficient [81].

2.2.2. *Presburger arithmetic.* The theory of *linear integer arithmetic* (LIA), also known as Presburger arithmetic, has the same syntax for formulas, but another semantics, replacing rational numbers ($\mathbb{Q}$) by integers ($\mathbb{Z}$). Linear inequalities are then insufficient for quantifier elimination — we also need congruence constraints: $\exists k\ x = 2k$ simplifies to $x \equiv 0 \pmod 2$.

Decision of formulas in Presburger arithmetic is doubly exponential, and thus quantifier elimination is very expensive in the worst case [48]. Presburger [89] provided a quantifier elimination procedure, but its complexity was impractical; Cooper [30] proposed a better algorithm [19, §7.2]; Pugh [90] proposed the "Omega test" [66, §5.5].

The practical complexity of Presburger arithmetic is high. In particular, the formulas produced tend to be very complex, even when there exists a considerably simpler and "understandable" equivalent formula, as seen with experiments in §7.1.

Cooper and Pugh's procedures are very geometrical in nature. Integers, however, can also be seen as words over the $\{0, 1\}$ alphabet, and sets of integers can thus be recognised by finite automata [88, §8]. Addition is encoded as a 3-track automaton recognising that the number on the third track truly is the sum of the numbers on the first two tracks; equivalently, this encodes subtraction. Existential quantifier elimination just removes some of the tracks, making transitions depending on bits read on that track nondeterministic. Negation is complementation (which can be costly, thus explaining the high cost of quantifier alternation). Multiplication by powers of two is also easily encoded, and multiplications by arbitrary constants can be encoded by a combination of additions and multiplications by powers of two.[5] The same idea can be extended to real numbers written as their binary expansion, using automata on infinite words.

This leads to an interesting arithmetic theory, with two sorts of variables: reals (or rationals) and integers. This could be used to model computer programs, with integers for integer variables and reals for floating-point variables (if necessary by using the semantic transformations described in § 4.5). Boigelot et al. [16] described a restricted class of ω-automata sufficient for quantifier elimination. Becker et al. [11] implemented the Lira tool based on such ideas. Unfortunately, this approach suffers from two major drawbacks: the practical performances are very bad for purely real problems [81], and it is impossible to recover an arithmetical formula from almost all these automata. We therefore did not pursue this direction.

2.2.3. *Nonlinear real arithmetic.* What happens if we do not limit ourselves to linear arithmetic, but also allow polynomials? Over the integers, the resulting theory is known as Peano arithmetic. It is well known that there can exist no decision procedure for quantified Peano arithmetic formulas.[6] Since a quantifier elimination algorithm would turn a quantified formula without free variables into an equivalent ground formula, and ground formulas

[5]This method embeds Presburger arithmetic into a stronger arithmetic theory, represented by the automata, then performs elimination over these automata. This partly explains why it is difficult to recover a Presburger formula from the resulting automaton.

[6]One does not need the full language of quantified Peano formulas for the problem to become undecidable. It is known that there exists no algorithm that decides whether a given nonlinear Diophantine equation (a polynomial equation with integer coefficients) has solutions, and that deciding such a problem is equivalent to deciding Turing's halting problem. in other words, it is impossible to decide whether a formula $P(x_1, \dots, x_n) = 0 \wedge x_1 \geq 0 \wedge \dots \wedge x_n \geq 0$ is satisfiable over the integers. See the literature on Hilbert's tenth problem, e.g. the book by Matiyasevich [73].

are trivially decidable, it follows that there can exist no quantifier elimination algorithm for this theory.

The situation is wholly different over the real numbers. The satisfiability or equivalence of polynomial formulas does not change whether the models are taken over the real numbers, the real algebraic numbers, or, for the matter, any *real closed field*; this theory is thus known as the theory of real closed fields, or *elementary algebra*. Tarski [105, 106] and Seidenberg [101] showed that this theory admits quantifier elimination, but their algorithms had impractically high complexity. Collins [28] introduced a better algorithm based on *cylindrical algebraic decomposition*. For instance, quantifier elimination on $\exists x\ ax^2+bx+c=0$ by cylindrical algebraic decomposition yields

$$\left(c<0\wedge\left(\left(b<0\wedge a\geq\frac{b^2}{4c}\right)\vee\ (b=0\wedge a>0)\vee\left(b>0\wedge a\geq\frac{b^2}{4\ c}\right)\right)\right)\vee c=0\vee \\ \left(c>0\wedge\left(\left(b<0\wedge\ a\leq\frac{b^2}{4c}\right)\vee(b=0\wedge a<0)\vee\left(b>0\wedge a\leq\ \frac{b^2}{4c}\right)\right)\right) \quad (2.1)$$

Note the cylindrical decomposition: first, there is a case disjunction according to the values of c, then, for each disjunct for c, a case disjunction for the value of b; more generally, cylindrical algebraic decomposition builds a tree of case disjunctions over a sequence of variables $v_1, v_2, \dots$, where the guard expressions defining the cases for v_i can only refer to $v_1, \dots, v_i$. This decomposition only depends on the polynomials inside the formula and not on its Boolean structure, and computing it may be very costly even if the final result is simple. This is the intuition why despite various improvements [10, 24] the practical complexity of quantifier elimination algorithms for the theory of real closed fields remain high. The theoretical space complexity is doubly exponential [21, 40]. This is why our results in §7.2 are of a theoretical rather than practical interest.

Minimal extensions to this formula language may lead to undecidability. This is for instance the case when one adds trigonometric functions: it is possible to define π as the least positive zero of the sine, then define the set of integers as the numbers k such that $\sin(k\pi)=0$, and thus one can encode Peano arithmetic formulas into that language [2]. Also, naive restrictions of the language do not lead to lower complexity. For instance, limiting the degree of the polynomials to two does not make the problem simpler, since formulas with polynomials of arbitrary degrees can be encoded as formulas with polynomials of degree at most two, simply by introducing new variables standing for subterms of the original polynomials. For instance, $\exists x\ ax^3+bx^2+cx+d=0$ can be encoded, using Horner's form for the polynomial, as $\exists x\exists y\exists z\ z=ax+b\wedge y=zx+c\wedge yx+d=0$. Certain stronger restrictions may however work; for instance, if the variables to be eliminated occur only linearly, then one can adapt the substitution methods described in §2.2.1.

3. Optimal Abstraction over Template Linear Constraint Domains

When applying abstract interpretation over domains of linear constraints, such as octagons [74, 76, 77], one generally applies a widening operator, which may lead to imprecisions. In some cases, *acceleration* techniques leading to precise results can be applied [52, 53]: instead of attempting to extrapolate a sequence of iterates to its limit, as does widening, the exact limit is computed. In this section, we describe a class of constraint domains and programs for which abstract transfer functions of loop-free codes and of loops can be exactly computed; thus the *optimality*. Furthermore, the analysis outputs these functions in closed form (as explicit expressions combining linear expressions and functional if-then-else constructs), so the result of the analysis of a program fragment can be stored away for future use; thus the *modularity.* Our algorithms are based on quantifier elimination over the theory of real linear arithmetic (§2.2).

3.1. **Template Linear Constraint Domains.** Let F be a formula over linear inequalities. We call F a domain definition formula if the free variables of F split into n *parameters* $p_1, \dots, p_n$ and m *state variables* $s_1, \dots, s_m$. We note $\gamma_F : \mathbb{Q}^n \to \mathcal{P}(\mathbb{Q}^m)$ defined by $\gamma_F(\vec{p}) = \{\vec{s} \in \mathbb{Q}^m \mid (\vec{p}, \vec{s}) \models F\}$. As an example, the interval abstract domain for 3 program variables s_1, s_2, s_3 uses 6 parameters $m_1, M_1, m_2, M_2, m_3, M_3$; the formula is $m_1 \le s_1 \le M_1 \wedge m_2 \le s_2 \le M_2 \wedge m_3 \le s_3 \le M_3$.

In this section, we focus on the case where F is a conjunction $L_1(s_1, \dots, s_m) \le p_1 \wedge \dots \wedge L_n(s_1, \dots, s_m) \le p_n$ of linear inequalities whose left-hand side is fixed and the right-hand sides are parameters. Such conjunctions define the class of *template linear constraint domains* [99]. Particular examples of abstract domains in this class are:

- the intervals (for any variable s, consider the linear forms s and $-s$); because of the inconvenience of talking intervals of the form $[-a, b]$, we shall often taking them of the form $[x_{\min}, x_{\max}]$, with the optimal value for $x_{\min}$ being a greatest lower bound instead of a least upper bound;
- the difference bound matrices (for any variables s_1 and s_2, consider the linear form $s_1 - s_2$);
- the octagon abstract domain (for any variables s_1 and s_2, distinct or not, consider the linear forms $\pm s_1 \pm s_2$) [74]
- the octahedra (for any tuple of variables $s_1, \dots, s_n$, consider the linear forms $\pm s_1 \cdots \pm s_n$). [25]

Remark that γ_F is in general not injective, and thus one should distinguish the *syntax* of the values of the abstract domain (the vector of parameters $\vec{p}$) and their *semantics* $\gamma_F(\vec{p})$. As an example, if one takes F to be $s_1 \le p_1 \wedge s_2 \le p_2 \wedge s_1 + s_2 \le p_3$, then both $(p_1, p_2, p_3) = (1, 1, 2)$ and $(1, 1, 3)$ define the same set for state variables s_1 and s_2. If $\vec{u} \le \vec{v}$ coordinate-wise, then $\gamma_F(\vec{u}) \subseteq \gamma_F(\vec{v})$, but the converse is not true due to the non-uniqueness of the syntactic form.

Take any nonempty set of states $W \subseteq \mathbb{Q}^m$. Take for all $i = 1, \dots, m$: $p_i = \sup_{\vec{s} \in W} L_i(\vec{s})$. Clearly, $W \subseteq \gamma_F(p_1, \dots, p_m)$, and in fact $\vec{p}$ is such that $\gamma_F(\vec{p})$ is the least solution to this inclusion. p_i belongs in general to $\mathbb{R} \cup \{+\infty\}$, not necessarily to $\mathbb{Q} \cup \{+\infty\}$. (for instance, if $W = \{s_1 \mid s_1^2 \le 2\}$ and $L_1 = s_1$, then $p_1 = \sqrt{2}$). We have therefore defined a map $\alpha_F : \mathcal{P}(\mathbb{R}^m) \to \{\bot\} \cup (\mathbb{R} \cup \{+\infty\})^n$, and (α_F, γ_F) form a *Galois connection*: α_F maps any set to its best upper-approximation.[7] The fixed points of $\alpha_F \circ \gamma_F$ are the *normal forms*; the normal form of $x^\sharp$ is the minimal abstract element that stands for the same

[7]See e.g. [35] for more information on Galois connection and their use in static analysis. Not all abstract interpretation techniques can be expressed within Galois connections. Indeed, there are abstract domains where there is not necessarily a best abstraction of a set of concrete states, e.g. the domain of convex polyhedra, which has no best abstraction for a disc, for instance. In this article, all abstractions, except the non-convex ones of § 4.3, are through Galois connections.

concrete set as $x^\sharp$.[8] For instance, $s_1 \leq 1 \wedge s_2 \leq 1 \wedge s_1 + s_2 \leq 2$ is in normal form, while $s_1 \leq 1 \wedge s_2 \leq 1 \wedge s_1 + s_2 \leq 3$ is not.

3.2. **Optimal Abstract Transformers for Program Semantics.** We shall consider the input-output relationships of programs with rational or real variables. We first narrow the problem to programs without loops and consider programs built from linear arithmetic assignments, linear tests, and sequential composition. Noting $a, b, \dots$ the values of program variables `a`, `b` ... at the beginning of execution and $a', b', \dots$ the output values, the semantics of a program P is defined as a formula $[\![P]\!]$ such that $(a, b, \dots, a', b', \dots) \models P$ if and only if the memory state $(a', b', \dots)$ can be reached at the end of an execution starting in memory state $(a, b, \dots)$:

Arithmetic: $[\![a := L(a, b, \dots) + K]\!]_F \triangleq a' = L(a, b, \dots) + K \wedge b' = b \wedge c' = c \wedge \dots$ where K is a real (in practice, rational) constant and L is a linear form, and $b, c, d \dots$ are all the variables except a;

Tests: $[\![\texttt{if } c \texttt{ then } p_1 \texttt{ else } p_2]\!] \triangleq (c \wedge [\![p_1]\!]_F) \vee (\neg c \wedge [\![p_2]\!]_F)$;

Non deterministic choice: $[\![a := \texttt{random}]\!] \triangleq b' = b \wedge c' = c \wedge \dots$, for all variables except a;

Failure: $[\![\texttt{fail}]\!] \triangleq \mathsf{false}$;

Skip: $[\![\texttt{skip}]\!] \triangleq a' = a \wedge b' = b \wedge c' = c \wedge \dots$

Sequence: $[\![P_1; P_2]\!]_F \triangleq \exists a'', b'', \dots\ f_1 \wedge f_2$ where f_1 is $[\![P_1]\!]_F$ where a' has been replaced by a'', b' by b'' etc., f_2 is $[\![P_2]\!]_F$ where a has been replaced by a'', b by b'' etc.

In addition to linear inequalities and conjunctions, such formulas contain disjunctions (due to tests and multiple branches) and existential quantifiers (due to sequential composition).

Note that so far, we have represented the concrete denotational semantics *exactly*. This representation of the transition relation using existentially quantified formulas is evidently as expressive as a representation by a disjunction of convex polyhedra (the latter can be obtained from the former by quantifier elimination and conversion to disjunctive normal form), but is more compact in general.

Consider now a domain definition formula $F \triangleq L_1(s_1, s_2, \dots) \leq p_1 \wedge \dots \wedge L_n(s_1, s_2, \dots) \leq p_n$ on the program inputs, with parameters $\vec{p}$ and free variables $\vec{s}$, and another $F' \triangleq L'_1(s'_1, s'_2, \dots) \leq p'_1 \wedge \dots \wedge L'_n(s'_1, s'_2, \dots) \leq p'_{n'}$ on the program outputs, with parameters $\vec{p'}$ and free variables $\vec{s'}$. Sound forward program analysis consists in deriving a *safe post-condition* from a precondition: starting from any state verifying the precondition, one should end up in the post-condition. Using our notations, the *soundness condition* is written

$$\forall \vec{s}, \vec{s'}\ F \wedge [\![P]\!] \implies F' \tag{3.1}$$

The free variables of this relation are $\vec{p}$ and $\vec{p'}$: the formula links the value of the parameters of the input constraints to admissible values of the parameters for the output constraints. Note that this soundness condition can be written as a universally quantified formula, with

[8] In the terminology of some authors, these can be referred to as the *reduced forms* or *closed forms*, and the $\alpha_F \circ \gamma_F$ operation is a *reduction* or *closure*. For instance, in the octagon abstract domain, the closure $\alpha_F \circ \gamma_F$ is implemented by a variant of Floyd-Warshall shortest path [74, 77].

no quantifier alternation. Alternatively, it can be written as a conjunction of correctness conditions for each output constraint parameter:

$$C'_i \triangleq \forall \vec{s}, \vec{s'}\ F \wedge [\![P]\!] \implies L'_i(\vec{s'}) \leq p'_i. \tag{3.2}$$

Let us take a simple example: if P is the program instruction $z := x + y$, $F \triangleq x \leq p_1 \wedge y \leq p_2$, $F' \triangleq z \leq p'_1$, then $[\![P]\!] \triangleq z' = x+y$, and the soundness condition is $\forall x, y, z'\ (x \leq p_1 \wedge y \leq p_2 \wedge z' = x + y \implies z' \leq p'_1)$. Remark that this soundness condition is equivalent to a formula without quantifiers $p'_1 \geq p_1 + p_2$, which may be obtained through quantifier elimination. Remark also that while any value for p'_1 fulfilling this condition is *sound* (for instance, $p'_1 = 1000$ for $p_1 = p_2 = 1$), only one value is *optimal* ($p'_1 = 2$ for $p_1 = p_2 = 1$). An optimal value for the output parameter p'_i is defined by $O'_i \triangleq C'_i \wedge \forall q'_i\ (C'_i[q'_i/p'_i] \implies p'_i \leq q'_i)$. Again, quantifier elimination can be applied; on our simple example, it yields $p'_1 = p_1 + p_2$.

If there are n input constraint parameters $p_1, \ldots, p_n$, then the optimal value for each output constraint parameter p'_i is defined by a formula O'_i with $n+1$ free variables $p_1, \ldots, p_n, p'_i$:

$$O'_i \triangleq C'_i \wedge \forall p''_i (C'_i[p''_i/p'_i] \Rightarrow p'_i \leq p'_i) \tag{3.3}$$

Lemma 1. The formula O'_i defined at Eq. 3.3, using the correctness subformula C'_i from Eq. 3.2, defines p'_i as the least possible value for the parameter of the constraint L_i after executing the transition $[\![p]\!]$ from a state verifying constraints F.

Proof. O'_i explicitly defines the least possible value of C'_i. C'_i explicitly defines all the acceptable values for parameter p'_i in the postcondition constraint. □

This formula defines a *partial function* from $\mathbb{Q}^n$ to $\mathbb{Q}$, in the mathematical sense: for each choice of $p_1, \ldots, p_n$, there exist at most a single p'_i. The values of $p_1, \ldots, p_n$ for which there exists a corresponding p'_i make up the *domain of validity* of the abstract transfer function. Indeed, this function is in general not defined everywhere; consider for instance the program:

```
if (x >= 10) { y = nondeterministic_choice_in_all_reals; }
else { y = 0; }
```

If $F = x \leq p_1$ and $F' = y \leq p'_1$, then $O'_1 \equiv p_1 < 10 \wedge p'_1 = 0$, and the function is defined only for $p_1 < 10$, with constant value 0.

At this point, we have a characterisation of the optimal abstract transformer corresponding to a program fragment P and the input and output domain definition formulas as n formulas (where n is the number of output parameters) O'_i each defining a function (in the mathematical sense) mapping the input parameters $\vec{p}$ to the output parameter p'_i.

Another example: the absolute value function $y := |x|$, again with the interval abstract domain. The semantics of the operation is $(x \geq 0 \wedge y = x) \vee (x < 0 \wedge y = -x)$; the precondition is $x \in [x_{\min}, x_{\max}]$ and the post-condition is $y \in [y_{\min}, y_{\max}]$. Acceptable values for $(y_{\min}, y_{\max})$ are characterised by formula

$$C' \triangleq \forall x\ x_{\min} \leq x \leq x_{\max} \implies y_{\min} \leq |x| \leq y_{\max} \tag{3.4}$$

The optimal value for $y_{\max}$ is defined by $O' \triangleq C' \wedge \forall y'_{\max}\ C'[y'_{\max}/y_{\max}] \implies y_{\max} \leq y'_{\max}$. Quantifier elimination over this last formula gives as characterisation for the least, optimal, value for $y_{\max}$:

$$(x_{\min} + x_{\max} \geq 0 \wedge y_{\max} = x_{\max}) \vee (x_{\min} + x_{\max} < 0 \wedge y_{\max} = -x_{\min}). \tag{3.5}$$

Listing 7: Optimal transformer for $y_{\max}$, corresponding to program $y = |x|$ with $x_{\min} \leq x \leq x_{\max}$

```
if (xmin + xmax >= 0) {
  ymax = xmax;
} else {
  ymax = -xmin;
}
```

We shall see in the next sub-section that such a formula can be automatically compiled into code (Listing 7).

3.3. Generation of the Implementation of the Abstract Domain.

Consider formula 3.5, defining an abstract transfer function. On this disjunctive normal form we see that the function we have defined is *piecewise linear*: several regions of the range of the input parameters are distinguished (here, $x_{\min} + x_{\max} < 0$ and $x_{\min} + x_{\max} \geq 0$), and on each of these regions, the output parameter $y_{\max}$ is a linear function of the input parameters. Given a disjunct (such as $y_{\max} = -x_{\min} \wedge x_{\min} + x_{\max} < 0$), the domain of validity of the disjunct can be obtained by existential quantifier elimination over the result variable (here $\exists y_{\max}\ (y_{\max} = -x_{\min} \wedge x_{\min} + x_{\max} < 0)$). The union of the domains of validity of the disjuncts is the domain of validity of the full formula. The domains of validity of distinct disjuncts can overlap, but in this case, since O'_i defines a partial function in the mathematical sense, that is, a relation $R(x, y)$ such that for any x there is at most one y such that $R(x, y)$, the functions defined by such disjuncts coincide on their overlapping domains of validity.

This suggests a first algorithm for conversion to an executable form:

(1) Put O'_i into quantifier-free, disjunctive normal form $C_1 \vee \cdots \vee C_n$.
(2) For each disjunct C_j, obtain the validity domain V_j as a conjunction of linear inequalities; this corresponds to a projection of the polyhedron defined by C_j onto variables $p_1, \ldots, p_n$, parallel to p'_i. Then, solve C_j for p'_i (obtain p'_i as a linear function v_i of the $p_1, \ldots, p_n$).
(3) Output the result as a cascade of if-then-else and assignments.

Consider now the example at the end of §3.2 and especially Formula 3.5, defining $y_{\max}$ as a function of $x_{\min}$ and $x_{\max}$: $(x_{\min} + x_{\max} \geq 0 \wedge y_{\max} = x_{\max}) \vee (x_{\min} + x_{\max} < 0 \wedge y_{\max} = -x_{\min})$. Let us first take the first disjunct $C_1 \triangleq x_{\min} + x_{\max} \geq 0 \wedge y_{\max} = x_{\max}$. Its validity domain is $\exists y_{\max}\ C_1$, that is, $x_{\min} + x_{\max} \geq 0$. Furthermore, on this validity domain, we can solve for $y_{\max}$ as a function of $x_{\min}$ and $x_{\max}$, and we obtain $y_{\max} = x_{\max}$. We therefore can print out the following test:

```
if (xmin + xmax >= 0) {
  ymax = xmax;
}
```

Now take the second disjunct $C_2 \triangleq (x_{\min} + x_{\max} < 0 \wedge y_{\max} = -x_{\min})$. It can similarly be turned into another test, which we put in the "else" branch of the preceding one. We thus obtain:

```
if (xmin + xmax >= 0) {
  ymax = xmax;
} else
if (xmin + xmax < 0) {
  ymax = -xmax;
}
```

Observe that in the above program, the second test is redundant. If we had been more clever, we would have realized that in the "else" branch of the first test, $x_{\min} + x_{\max} < 0$ always holds, and thus it is useless to test this condition. Furthermore, in an if-then-else cascade obtained with the above method, the same condition may have to be tested several times. We shall now propose an algorithm that constructs an if-then-else *tree* such that no useless tests are generated.

Algorithm 1 ToITEtree(F, p', T): turn a formula defining p' as a function of the other free variables of F into a tree of if-then-else constructs, assuming that T holds.

```
  $D(= C_1 \vee \cdots \vee C_n) \leftarrow$ QElimDNFModulo($\{\}, F, T$)
  for all $C_i \in D$ do
    $P_i \leftarrow$ QElimDNFModulo($p', F, T$)
  end for
  $P \leftarrow$ Predicates($P_1, \dots, P_n$)
  if $P = \emptyset$ then
Ensure:   $\exists p'\ F$ is always true
    $O \leftarrow$ Solve($D, p'$)
  else
    $K \leftarrow$ Choose($P$)
    $O \leftarrow$ IfThenElse($K$, ToITEtree($F, p', T \wedge K$), ToITEtree($F, p', T \wedge \neg K$))
  end if
```

The idea of the algorithm is as follows:

- Each path in the if-then-else tree corresponds to a conjunction C of conditions (if one goes through the "if" branch of `if (a)` and the "else" branch of `if (b)`, then the path corresponds to $a \wedge \neg b$).
- The formula O'_i is simplified relatively to C, a process that prunes out conditions that are always or never satisfied when C holds.
- If the path is deep enough, then the simplified formula becomes a conjunction. This conjunction, as a relation, is a partial function from the $p_1, \dots, p_n$ to the p'_i we wish to compute. Again, we solve this conjunction to obtain p'_i as an explicit function of the $p_1, \dots, p_n$. For instance, in the above example, we obtain $y_{\max}$ as a function of $x_{\min}$ and $x_{\max}$.

We say that two formulas A and B are equivalent, denoted by $A \equiv B$, if they have the same models, and we say that they are equivalent modulo a third formula T, denoted by $A \equiv_T B$, if $A \wedge T \equiv B \wedge T$. Intuitively, this means that we do not care what happens where F is false, thus the terminology "don't care set" sometimes found for $\neg F$.

QElimDNFModulo($\vec{v}, F, T$) is a function that, given a possibly empty vector of variables $\vec{v}$, a formula F and a formula T, outputs a quantifier-free formula F' in disjunctive normal form such that $F' \equiv_T \exists \vec{v}\ F$ and no useless predicates are used. In [81], we have presented such a function as a variant of quantifier elimination.

We need some more auxiliary functions. $\textsc{Predicates}(F)$ returns the set of atomic predicates of F. $\textsc{Solve}(C, p')$ solves a conjunction of inequalities C for variable p'. If C does not contain redundant inequalities, then it is sufficient to look for inequalities of the form $p' \geq L$ or $p' \leq L$ and output $p' = L$.[9] Finally $\textsc{Choose}(P)$ chooses any predicate in the set of predicates P; one good heuristic seems to be to choose the most frequent atomic predicate where p'_i does not occur.

Let us take, as a simple example, Formula 3.5. We wish to obtain $y_{\max}$ as a function of $x_{\min}$ and $x_{\max}$, so in the algorithm $\textsc{ToITEtree}$ we set $p' \triangleq y_{\max}$. C_1 is the first disjunct $x_{\min} + x_{\max} \geq 0 \wedge y_{\max} = x_{\max}$, C_2 is the second disjunct $x_{\min} + x_{\max} < 0 \wedge y_{\max} = -x_{\min}$. We project C_1 and C_2 parallel to $y_{\max}$, obtaining respectively $P_1 = (x_{\min} + x_{\max} \geq 0)$ and $P_2 = (x_{\min} + x_{\max} < 0)$. We choose K to be the predicate $x_{\min} + x_{\max} \geq 0$ (in this case, the choice does not matter, since P_1 and P_2 are the negation of each other).

- The first recursive call to $\textsc{ToITEtree}$ is made with $T \triangleq (x_{\min} + x_{\max} \geq 0)$. Obviously, $F \wedge T \equiv (y_{\max} = x_{\max}) \wedge T$ and thus $(\exists y_{\max} F) \wedge T \equiv T$.

  $\textsc{QElimDNFModulo}(y_{\max}, F, T)$ will then simply output the formula "true". It then suffices to solve for $y_{\max}$ in $y_{\max} = x_{\max}$. This yields the formula for computing the correct value of $y_{\max}$ in the cases where $x_{\min} + x_{\max} \geq 0$.
- The second recursive call is made with $T \triangleq (x_{\min} + x_{\max} < 0$. The result is $y_{\max} = -x_{\min}$, the formula for computing the correct value of $y_{\max}$ in the cases where $x_{\min} + x_{\max} < 0$.

These two results are then reassembled into a single if-then-else statement, yielding the program at the end of §3.2.

The algorithm terminates because paths of depth d in the tree of recursive calls correspond to truth assignments to d atomic predicates among those found in the domains of validity of the elements of the disjunctive normal form of F. Since there is only a finite number of such predicates, d cannot exceed that number. A single predicate cannot be assigned truth values twice along the same path because the simplification process in $\textsc{QElimDNFModulo}$ erases this predicate from the formula.

This algorithm seems somewhat unnecessarily complex. It is possible that techniques based upon AIGs, performing simplification with respect to "don't care sets" [100], could also be used.

3.4. **Least Inductive Invariants.**

We have so far considered programs without loops. The analysis of program loops, as well as proofs of correctness of programs with loops using Floyd-Hoare semantics, is based upon the notion of *inductive invariants.* A set of states I is deemed to be an inductive invariant for a loop if it contains the initial state and it is stable by the loop iteration — in other words, if it is impossible to move from a state inside I to a state outside I by one iteration of the loop. The intersection of all inductive invariants is also an inductive invariant — in fact, it is the least inductive invariant.

Any property true over the least inductive invariant of a loop is true throughout the execution of that loop; for this reason, some authors call such properties *invariants* (without

[9]In practice, any library for convex polyhedra can provide a basis of the equalities implied by a polyhedron given by constraints, in other words a system of linear inequalities defining the affine span of the polyhedron. It is therefore sufficient to take that basis and keep any equality where p' appears.

the "inductive" qualifier), while some other authors call invariants what we call inductive invariants.

It would be interesting to be able to compute the least invariant (inductive or noninductive) within our chosen abstract domain; in other words, compute the least element in our abstract domain that contains the least inductive invariant of the loop or program. Unfortunately, this is in general impossible; indeed, doing so would entail solving the halting problem. Just take any program P, create a fresh variable, and consider the program that initialises x to 0, runs P, and then sets x to 1. Clearly, the least invariant interval for this program and variable x is $[0,0]$ if P does not terminate, and $[0,1]$ if it does terminate.

We thus settle for a simpler problem: find the *least inductive invariant within our abstract domain*, that is, the least element in our abstract domain that is an inductive invariant. Note that even on very simple examples, this can be vastly different from computing the least invariant within the abstract domain. Take for instance the simple program of Fig. 5, which has a couple of real variables (x, y) and rotates them by $45°$ at each iteration. The (x, y) couple always stays within the square $[-1,1] \times [-1,1]$, so this square is an invariant within the interval domain. Yet this square is not an inductive invariant, for it is not stable if rotated by $45°$; in fact, the only inductive invariant within the interval domain is $(-\infty,+\infty)\times(-\infty,+\infty)$, which is rather uninteresting! Note that on the same figure, the octagon abstract domain would succeed (and produce a regular octagon centered on $(0,0)$).

3.4.1. *Stability Inequalities.* Consider a program fragment: **while** (c) { p; }. We have domain definition formulas $F \triangleq L_1(s_1,\dots,s_m) \le p_1 \wedge \dots \wedge L_n(s_1,\dots,s_m) \le p_n$ for the precondition of the program fragment , and $F' \triangleq L'_1(s_1,\dots,s_m) \le p'_1 \wedge \dots \wedge L'_n(s_1,\dots,s_m) \le p'_{n'}$ for the invariant.

Define $G = [\![c]\!] \wedge [\![p]\!]$. G is a formula whose free variables are $s_1,\dots,s_m,s'_1,\dots,s'_m$ such that $(s_1,\dots,s_m,s'_1,\dots,s'_m) \models G$ if and only if the state $(s'_1,\dots,s'_m)$ can be reached from the state $(s_1,\dots,s_m)$ in exactly one iteration of the loop. A set $W \subseteq \mathbb{Q}^m$ is said to be an *inductive invariant* for the head of the loop if $\forall \vec{s} \in W, \forall \vec{s'}\ (\vec{s},\vec{s'}) \models G \implies \vec{s'} \in W$. We seek inductive invariants of the shape defined by F', thus solutions for $\vec{p'}$ of the *stability condition*:

$$\forall \vec{s}, \vec{s'}\ F' \wedge G \implies F'[\vec{s'}/\vec{s}]. \tag{3.6}$$

Not only do we want an inductive invariant, but we also want the initial states of the program to be included in it. The condition then becomes

$$H \triangleq (\forall \vec{s}, F \implies F') \wedge (\forall \vec{s}, \vec{s'}\ F' \wedge G \implies F'[\vec{s'}/\vec{s}]) \tag{3.7}$$

This is an invariant condition for the head A of a loop written as:

```
loop:
  /* A */
  if (! c) goto end;
  /* B */
  loop body
  goto loop;
end:
```

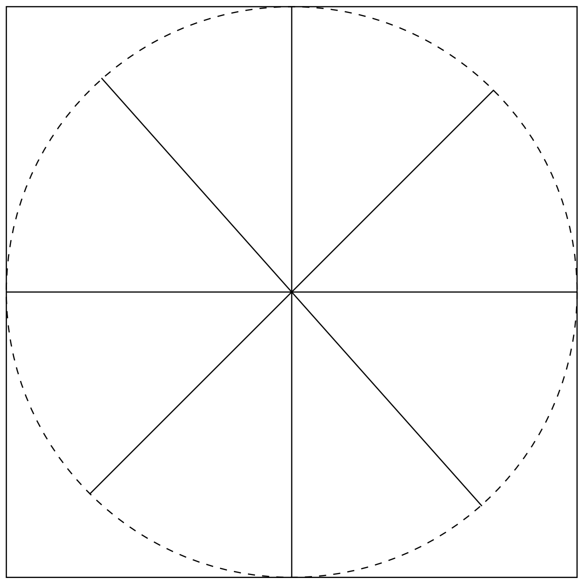

Figure 5: The least fixed point representable in the domain ($\text{lfp}(\alpha \circ f \circ \gamma)$) is not necessarily the least approximation of the least fixed point ($\alpha(\text{lfp}\, f)$) inside the abstract domain. For instance, if we take a program initialised by $x \in [-1, 1]$ and $y = 0$, and at each iteration, we rotate the point by $45°$, the least invariant is an 8-point star, and its best approximation inside the abstract domain of intervals is the square $[-1, 1]^2$. However, this square is not an inductive invariant: no rectangle (product of intervals) is stable under the iterations, thus there is no abstract inductive invariant within the interval domain. Using the domain of convex polyhedra, one would obtain a regular octagon.

Alternatively, one can consider an invariant condition for location B. The condition then becomes

$$H_{\text{alt}} \triangleq \forall \vec{s}\ [\![c]\!] \wedge (F \vee (\exists \vec{s''}\ [\![p]\!][\vec{s''}/\vec{s}, \vec{s}/\vec{s'}] \wedge F'[\vec{s''}/\vec{s}])) \implies F' \tag{3.8}$$

This alternate condition is very similar to the previous one (a universally quantified formula with no alternation, since the $\exists$ is negated). For the sake of simplicity, we shall only discuss the treatment of formula H; formula H_{alt} can be treated in the same way.

This formula links the values of the input constraint parameters $p_1, \ldots, p_n$ to acceptable values of the invariant constraint parameters $p'_1, \ldots, p'_{n'}$. In the same way that our soundness or correctness condition on abstract transformers allowed any sound post-condition, whether optimal or not, this formula allows any inductive invariant of the required shape as long as it contains the precondition, not just the least one.

The intersection of sets defined by $\vec{p'}_1$ and $\vec{p'}_2$ is defined by $\min(\vec{p'}_1, \vec{p'}_2)$. More generally, the intersection of a family of sets, unbounded yet closed under intersection, defined by $\vec{p'} \in Z$ is defined by $\min\{p' \mid p' \in Z\}$. We take for Z the set of acceptable parameters $\vec{p'}$ such that $\vec{p'}$ defines an inductive invariant and $\forall \vec{s}, F \implies F'$; that is, we consider only inductive invariants that contain the set $I = \{\vec{s} \mid F\}$ of precondition states.

We deduce that p'_i is uniquely defined by: $p'_i = \min(\exists p'_1, \dots, p'_{i-1}, p'_{i+1}, \dots, p'_{n'}\ H)$ which can be rewritten as

$$O'_i \triangleq (\exists p'_1, \dots, p'_{i-1}, p'_{i+1}, \dots, p'_{n'}\ H) \land (\forall \vec{q'}\ H[\vec{q'}/\vec{p'}] \implies p'_i \leq q'_i) \tag{3.9}$$

The free variables of this formula are $p_1, \dots, p_n, p'_i$. This formula defines a function (in the mathematical sense) defining p'_i from $p_1, \dots, p_n$. As before, this function can be compiled to an executable version using cascades or trees of tests.

Lemma 2. The formula O'_i defined at Eq. 3.9 defines the least value of p'_i for an inductive invariant of the shape defined by F for the transition relation defined by G.

Proof. Similarly as for lemma 1, the formula H defined at Eq. 3.7 defines a set Y of admissible $p'_1, \dots, p'_{n'}$ such that $F \triangleq L_1(s_1, \dots, s_m) \leq p'_1 \land \dots \land L_n(s_1, \dots, s_m) \leq p'_{n'}$ is an inductive invariant for the loop. Formula O'_i defines p'_i to be $\inf\{p''_i \mid (p''_1, \dots, p''_n) \in Y\}$. in other words, $(p'_1, \dots, p'_{n'}) = \inf Y$. $\square$

Thus the overall operation of the analysis method:

We start from quantified formulas O'_i defining the least inductive invariant of the loop *in the abstract domain* (Lemma 2). We eliminate quantifiers from these formulas; since this does not change their models, the resulting formulas without quantifiers also define the least inductive invariant in the abstract domain.

- Either the problem had no precondition parameters $p_1, \dots, p_n$, and thus each formula O'_i has only one variable p'_i. It consists of linear (in)equalities, and has a single model for p'_i, which is straightforward to extract. Collect these values, one obtains the values defining the least invariant $(p'_1, \dots, p'_{n'})$ in the abstract domain.
- Either the problem has precondition parameters and one employs one of the methods in §3.3 to obtain equivalent executable code.

The overall correctness of the method is quite tautological. We start from a non-executable specification of the least inductive invariant in the abstract domain in the form of quantified formulas. We eliminate the quantifiers from these formulas, then process them into equivalent executable code. At all steps, we have preserved logical equivalence with the original definition. In short, we have synthetized the implementation of the best transformer from its specification.

3.4.2. *Simple Loop Example.* To show how the method operates in practice, let us consider first a very simple example (nondet() is a nondeterministic choice):

```
int i=0;
while (i <= n) {
  if (nondet()) {
    i=i+1;
    if (i == n) {
      i=0;
    }
  }
}
```

Let us abstract `i` at the head of the loop using an interval $[i_{\min}, i_{\max}]$. For simplicity, we consider the case where the loop is at least entered once, and thus $i = 0$ belongs to the invariant. For better precision, we model each comparison $x \neq y$ over the integers as $x >= y + 1 \vee x <= y - 1$; similar transformations apply for other operators. The formula expressing that such an interval is an inductive invariant is:

$$i_{\min} \leq 0 \wedge 0 \leq i_{\max} \wedge \forall i \forall i' \; ((i_{\min} \leq i \wedge i \leq i_{\max} \wedge \\ (((i + 1 \leq n - 1 \vee i + 1 \geq n + 1) \wedge i' = i + 1) \vee \\ (i + 1 = n + 1 \wedge i' = 0) \vee i' = i)) \implies (i_{\min} \leq i' \wedge i' \leq i_{\max})) \quad (3.10)$$

Quantifier elimination produces:

$$(i_{\min} \leq 0 \wedge i_{\max} \geq 0 \wedge i_{\max} < n \wedge -i_{\min} + n - 2 < 0) \vee \\ (i_{\min} \leq 0 \wedge i_{\max} \geq 0 \wedge i_{\max} - n + 1 \geq 0 \wedge i_{\max} < n) \quad (3.11)$$

The formulas defining optimal $i_{\min}$ and $i_{\max}$ are:

$$i_{\min} \geq 0 \wedge i_{\min} \leq 0 \wedge n > 0 \quad (3.12)$$

$$(i_{\max} = 0 \wedge n > 0 \wedge n < 2) \vee (i_{\max} = n - 1 \wedge i_{\max} \geq 1) \quad (3.13)$$

We note that this invariant is only valid for $n > 0$, which is unsurprising given that we specifically looked for invariants containing the precondition $i = 0$. The output abstract transfer function is therefore:

```
if (n <= 0) {
  fail();
} else {
  iMin = 0;
  if (n < 2) {
    iMax = 0;
  } else /* n >= 2 */
    iMax = n-1;
  }
}
```

The case disjunction `n < 2` looks unnecessary, but is a side effect of the use of rational numbers to model a problem over the integers. The resulting abstract transfer function is optimal, but on such a simple case, one could have obtained the same using polyhedra [36] or octagons [74].

We have already noted (§2.1) that even with we replace `n` by the constant 10, the classical widening/narrowing approach will fail to identify the least invariant of this loop, and that extra techniques such have widening with thresholds have to be used.

3.4.3. *Synchronous Data Flow Example: Rate Limiter.* To go back to the original problem of floating-point data in data-flow languages, let us consider the following library block: a *rate limiter*. As seen in Listing 8, such a block in inserted in a reactive loop, as shown below, where **assume**(c) stands for **if** (c) {} **else** {fail ();} and fail () aborts execution.

This block has three input streams e1, e2, and e3, and one output stream s1. In intuitive terms, s1 is the same as e1 but where the maximal slope of the signal between two successive clock ticks is bounded by e2, thus the name *rate limiter*. At some points in time

Listing 8: Rate limiter.

```
while (true) {
  ...
  e1 = random(); assume(e1 >= e1min && e1 <= e1max);
  e2 = random(); assume(e2 >= e2min && e2 <= e2max);
  e3 = random(); assume(e3 >= e3min && e3 <= e3max);
  olds1 = s1;
  if (nondet()) {
    s1 = e3;
  } else {
    if (e1 - olds1 < -e2) {
      s1 = olds1 - e2;
    }
    if (e1 - olds1 > e2) {
      s1 = olds1 + e2;
    }
  }
  ...
}
```

Listing 9: If then else tree

```
if (e1max > e3max) {
  s1max = e1max;
} else {
  s1max = e3max;
}
```

(modelled by nondeterministic choice), the value of the signal is reset to that of the third input e3.

We are interested in the input-output behaviour of that block: obtain bounds on the output s1 of the system as functions of bounds on the inputs (e1, e2, e3). One difficulty is that the `s1` output is memorised, so as to be used as an input to the next computation step. The semantics of such a block is therefore expressed as a fixed point.

We wish to know the least inductive invariant of the form $s_{1\min} \leq s_1 \leq s_{1\max}$ under the assumption that $e_{1\min} \leq e_{1\max} \wedge e_{2\min} \leq e_{2\max} \wedge e_{3\min} \leq e_{3\max}$. The stability condition yields, after quantifier elimination and projection on $s_{1\max}$ of the condition $s_{1\max} \geq e_{1\max} \wedge s_{1\max} \geq e_{3\max}$. Minimisation then yields an expression that can be compiled to an if-then-else tree (Listing 9).

This result, automatically obtained, coincides with the intuition that a rate limiter (at least, one implemented with exact arithmetic) should not change the range of the signal that it processes.

This program fragment has a rather more complex behaviour if all variables and operations are IEEE-754 floating-point, since rounding errors introduce slight differences of regimes between ranges of inputs. Rounding errors in the program to be analysed introduce difficulties for analyses using widenings, since invariant candidates are likely to be “almost stable”, but not truly stable, because of these errors. Again, there exist workarounds so that widening-based approaches can still operate [15, Sec. 7.1.4]. We shall see in §4.5 how to correctly abstract floating-point behaviour within our framework; unfortunately, the formulas produced tend to be very large due to many case disjunctions. The implementation of the abstract transformer produced for the above rate limiter in floating-point does not fit within one page of article, this is why we omitted it.

4. Extensions of the framework using real linear arithmetic

We shall describe here a few extensions to the class of programs and domains that we can handle, all of which are still based on quantifier elimination over real linear arithmetic. (In §7, we shall investigate extensions using other arithmetic theories.)

4.1. Emptiness.

We have so far supposed that the statement where the inductive invariant is computed is reachable, and thus that there exists some nonempty set of initial states that constrain the inductive invariant from below. More generally, and especially for the constructs described in §4.4 and §5.1, it may be necessary to encode the bottom element $\perp$ of the abstract domain, which represents the empty set of states. This can be done using one Boolean variable per element that might be empty: instead of template parameters $(p_1, \dots, p_n)$, we will have $(b, p_1, \dots, p_n)$, with the semantics that $\gamma(\text{false}, p_1, \dots, p_n) = \emptyset$ and $\gamma(\text{true}, p_1, \dots, p_n) = \{\vec{s} \mid \forall i\ L_i(\vec{s}) \leq p_i\}$.

Sankaranarayanan et al. [99] use $p_i = -\infty$ for this, but in our framework, infinities themselves have to be encoded using Booleans, as we'll see in the next subsection. Furthermore, if we have an abstract element in normal form with a constraint $L_i(v_1, \dots) \leq -\infty$, it means that all p_j are $-\infty$ and thus it is sufficient to have a single Boolean variable for all of them.

4.2. Infinities.

The techniques explained in Sec. 3.1 allow only finite bounds. Consider for instance the following program:

```
x = 0;
while (true) {
  x = x+1;
}
```

We would like to obtain as a result of the analysis that x lies in $[0, \infty)$. Yet, what will happen is that the formula describing the couples $(x_{\min}, x_{\max})$ defining inductive invariants $x_{\min} \leq x \leq x_{\max}$ will be simplified to false.

More annoyingly, with the following program:

```
x = y;
while (true) {
  x = x+1;
}
```

we would at least hope to infer that $x_{\min} = y_{\min}$. Yet, if we look for an invariant of the form $x_{\min} \leq x \leq x_{\max}$, there is no solution for any value of $(y_{\min}, y_{\max})$! In §7.1 we shall see an example where it is actually interesting to infer the domain of values of the precondition where the least inductive invariant interval is finite, and that this domain can simply be obtained by existential quantification on the parameters of the inductive invariant followed by elimination. But in general, this is not what we want; instead, we would prefer to allow infinite values in the p and p'.

This can be easily achieved by a minor alteration to our definitions. Each parameter p_i is replaced by two parameters p_i^b and p_i^∞. p_i^∞ is constrained to be in $\{0, 1\}$ (if the quantifier elimination procedure in use allows Boolean variables, then p_i^∞ can be taken as a Boolean variable); $p_i^\infty = 0$ means that p_i is finite and equal to p_i^b, $p_i^\infty = 1$ means $p_i = +\infty$. $L_i \leq p_i$ becomes $(p_i^\infty > 0) \vee (L_i \leq p_i^b)$, $L_i < p_i$ becomes $(p_i^\infty > 0) \vee (L_i < p_i^b)$. After this rewriting, all formulas are formulas of the theory of linear inequalities without infinities and are amenable to the appropriate algorithms.

Unfortunately, the added combinatorial complexity induced by these Boolean variables tends to lead to intolerable computation times in the quantifier elimination procedures. Further work is needed, probably in the direction of better quantifier elimination procedures for combinations of Boolean and real quantified variables. Alternatively, one can envision directly including reasoning about infinities inside these procedures, though this is of course a delicate matter because of the possibility of generation of indeterminate forms $\infty - \infty$ if formulas are handled without special care.

4.3. **Non-Convex Domains.** Section 3.1 constrains formulas to be conjunctions of inequalities of the form $L_i \leq p_i$. What happens if we consider formulas that may contain disjunctions?

The template linear constraint domains of section 3.1 have a very important property: they are closed under (infinite) intersection; that is, if we have a family $\vec{p} \in W$, then there exists p_0 such that $\bigcap_{\vec{p} \in W} \gamma_F(\vec{p}) = \gamma_F(\vec{p_0})$ (besides, $p_0 = \inf\{\vec{p} \mid \vec{p} \in W\}$). This is what enables us to request the *least* element that contains the exact post-condition, or the least inductive invariant in the domain: we take the intersection of all acceptable elements.

Yet, if we allow non-convex domains, there does not necessarily exist a least element $\gamma_F(\vec{p})$ such that $S \subseteq \gamma_F(\vec{p})$. Consider for instance $S = \{0, 1, 2\}$ and F representing unions of two intervals $((-x \leq p_1 \wedge x \leq p_2) \vee (-x \leq p_3 \wedge x \leq p_4)) \wedge p_2 \leq -p_3$. There are two, incomparable, minimal elements of the form $\gamma_F(\vec{p})$: $p_1 = p_2 = 0 \wedge p_3 = -1 \wedge p_4 = 2$ and $p_1 = 0 \wedge p_2 = 1 \wedge p_3 = -2 \wedge p_4 = 2$.

We consider formulas F built out of linear inequalities $L_i(s_1, \ldots, s_n) \leq p_i$ as atoms, conjunctions, and disjunctions. By induction on the structure of F, we can show that $\gamma_F : (\mathbb{R} \cup \{-\infty\})^n \to \mathcal{P}(\mathbb{R}^n)$ is inf-continuous; that is, for any descending chain $(\vec{p_i})_{i \in I}$ such that $\lim_i \vec{p_i} = \vec{p_\infty}$, then $\gamma_F(\vec{p_i})$ is decreasing and $\bigcap_{i \in I} \gamma_F(\vec{p_i}) = \gamma_F(\vec{p_\infty})$. The property is trivial for atomic formulas, and is conserved by greatest lower bounds ($\wedge$) as well as binary least upper bounds ($\vee$).

Let us consider a set $S \subseteq \mathcal{P}(\mathbb{R}^n)$, stable under arbitrary intersection (or at least, greatest lower bounds of descending chains). S can be for instance the set of invariants of a relation, or the set of over-approximations of a set W. $\gamma_F^{-1}(S)$ is the set of suitable domain parameters; for instance, it is the set of parameters representing inductive invariants of the shape specified by F, or the set of representable over-approximations of W. $\gamma_F^{-1}(S)$ is stable under greatest lower bounds of descending chains: take a descending chain $(\vec{p_i})_{i \in I}$,

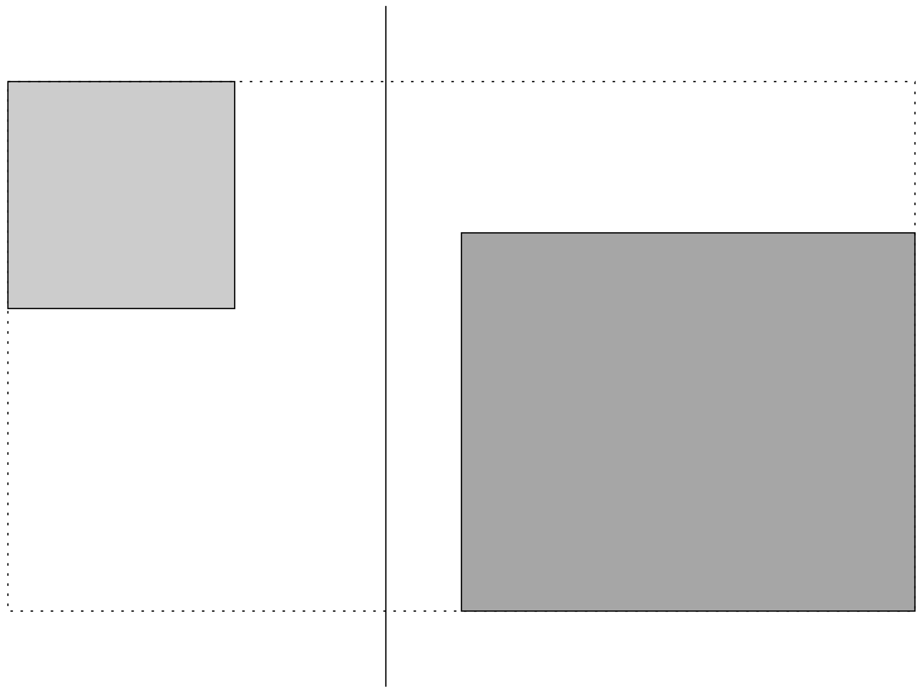

Figure 6: The state space is partitioned into $x < 0$ and $x \geq 0$, and on each element of the partition we have a product of intervals, respectively $[-5, -2] \times [4, 7]$ and $[1, 7] \times [0, 5]$. Without the disjunction, we would have had to consider the much larger set $[-5, 7] \times [0, 7]$.

then $\gamma_F(\lim_i \vec{p_i}) = \cap_i \gamma_F(\vec{p_i}) \in S$ by inf-continuity and stability of S. By Zorn's lemma, $\gamma_F^{-1}(S)$ has at least one minimal element.

Let $P[\vec{p}]$ be a formula representing $\gamma_F^{-1}(S)$ (Sec. 3.1 proposes formulas defining safe post-conditions and inductive invariants). The formula $G[\vec{p}] \triangleq P[\vec{p}] \land \forall \vec{p'}\ P[\vec{p'}] \land \vec{p'} \leq \vec{p} \implies \vec{p} \leq \vec{p'}$ defines the minimal elements of $\gamma^{-1}(S)$.

For instance, consider $\vec{p} = (a, b, c, d)$, $F \triangleq (-x \leq a \land x \leq b) \lor (-x \leq c \land x \leq d)$, representing unions of two intervals $[-a, b] \cup [-c, d]$. We want upper-approximations of the set $\{0, 1, 3\}$; that is $P[\vec{p}] \triangleq \forall x\ (x = 0 \lor x = 1 \lor x = 3 \implies F[\vec{p}, x])$. We add the constraint that $-a \leq b \land b \leq -c \land -c \leq d$, so as not to obtain the same solutions twice (by exchange of (a, b) and (c, d)) or solutions with empty intervals. By quantifier elimination over G, we obtain $(a = 0 \land b = 1 \land c = -3 \land d = 3) \lor (a = 0 \land b = 0 \land c = -1 \land d = 3)$, that is, either $[0, 0] \cup [1, 3]$ or $[0, 1] \cup [3, 3]$.

4.4. **Domain Partitioning.** Non-convex domains, in general, are not stable under intersections and thus "best abstraction" problems admit multiple solutions as minimal elements of the set of correct abstractions. As explained in the preceding subsection, is not very satisfactory nor efficient for analysis. There are, however, non-convex abstract domains that are stable under intersection and thus admit least elements as well as the template linear constraint domains of Sec. 3.1: those defined by partitioning of the state space.

If, for instance, we know that a program or system behaves differently according to the sign of x, then we can decide *in advance* to partition the state space into $x \geq 0$ and $x < 0$. On each element of the partition, we can have a separate template (example Fig. 6). We can accommodate this within our framework.

More formally, consider pairwise disjoint subsets $(C_i)_{i \in I}$ of the state space $\mathbb{Q}^m$, and abstract domains stable under intersection $(S_i)_{i \in I}$, $S_i \subseteq \mathcal{P}(C_i)$. Elements of the partitioned abstract domain are unions $\bigcup_{i \in I} s_i$ where $s_i \in S_i$. If $\left(\bigcup_i s_{i,j}]\right)_{j \in J}$ is a family of elements of the domain, then $\bigcap_{j \in J} \left(\bigcup_{i \in I} s_{i,j}\right) = \bigcup_{i \in I} \bigcap_{j \in J} s_{i,j}$; that is, intersections are taken separately in each C_i. in other words, the parameters of the templates on each element of the partition can be dealt with independently of each other.

Note the difference with the general disjunctive domains of §4.3: in the general disjunctive domains, there is no partition fixed *a priori*, this is why we may have several incomparable minimal elements $[0,0] \cup [1,2]$ and $[0,1] \cup [2,2]$ in the domain of disjunctions of two intervals representing the same set $\{0,1,2\}$. For a fixed partition C_i and corresponding domains S_i, for any set X, for every i, there is a least element representing $X \cap C_i$ in domain S_i. This motivates the following construct:

Take a family $(F_i[\vec{p}])_{i \in I}$ of formulas defining template linear constraint domains (conjunctions of linear inequalities $L_i(s_1, \dots, s_n) \leq p_i$) and a family $(C_i)_{i \in I}$ of formulas such that for all i and i', $C_i \wedge C_{i'}$ is equivalent to false and $C_1 \vee \dots \vee C_l$ is equivalent to true. $F = (C_1 \wedge F_1) \vee \dots \vee (C_l \wedge F_l)$ then defines an abstract domain such that γ_F is a inf-morphism. All the techniques of §3.1 then apply.

For instance, by choosing $C_1 \triangleq x \geq 0$, $C_2 \triangleq x < 0$, $F_1 \triangleq x^1_{\min} \leq x \leq x^1_{\max} \wedge y^1_{\min} \leq x \leq y^1_{\max}$, and $F_2 \triangleq x^2_{\min} \leq x \leq x^2_{\max} \wedge y^2_{\min} \leq x \leq y^2_{\max}$, we can obtain Figure 6.

The above constructions are equivalent to assigning a separate control point to each element in the partition, with guards leading to these points according to the C_i, and then performing as described in §5.1.

4.5. Floating-Point Computations.

Real-life programs do not operate on real numbers; they operate on fixed-point or floating-point numbers. Floating point operations have few of the good algebraic properties of real operations; yet, they constitute approximations of these real operations, and the *rounding error* introduced can be bounded.

In IEEE floating-point [61], each atomic operation (noting $\oplus$, $\ominus$, $\otimes$, $\oslash$, $\sqrt{}_f$ for operations so as to distinguish them from the operations $+$, $-$, $\times$, $/$, $\sqrt{}$ over the reals) is mathematically defined as the image of the exact operation over the reals by a rounding function.[10] This rounding function, depending on user choice, maps each real x to the nearest floating-point value $r_n(x)$ (*round to nearest mode*, with some resolution mechanism for non representable values exactly in the middle of two floating-point values), $r_{-\infty}(x)$ the greatest floating-point value less or equal to x (*round toward* $-\infty$), $r_{+\infty}(x)$ the least floating-point value greater or equal to x (*round toward* $+\infty$), $r_0(x)$ the floating-point value of the same sign as x but whose magnitude is the greatest floating-point value less or equal to $|x|$ (*round toward* 0). If x is too large to be representable, $r(x) = \pm\infty$ depending on the size of x

[10]We leave aside the peculiarities of some implementations, such as those of most C compilers over the 32-bit Intel platform where there are "extended precisions" types used for some temporary variables and expressions can undergo double rounding [84].

The semantics of the rounding operation cannot be exactly represented inside the theory of linear inequalities.[11] As a consequence, we are forced to use an axiomatic over-approximation of that semantics: a formula linking a real number x to its rounded version $r(x)$.

Miné [75] uses an inequality $|r(x) - x| \leq \varepsilon_{\text{rel}} \cdot |x| + \varepsilon_{\text{abs}}$, where ε_{rel} is a *relative error* and ε_{abs} is an *absolute error*, leaving aside the problem of overflows. The relative error is due to rounding at the last binary digit of the significand, while the *absolute error* is due to the fact that the range of exponents is finite and thus that there exists a least positive floating-point number and some nonzero values get rounded to zero instead of incurring a relative error (or get rounded to a denormal, see below).

Because our language for axioms is richer than the interval linear forms used by Miné, we can express more precise properties of floating-point rounding. We recall briefly the characteristics of IEEE-754 floating-point numbers. Nonzero floating point numbers are represented as follows: $x = \pm 2^e m$ where $1 \leq m < 2$ is the *mantissa* or *significand*, which has a fixed number p of bits, and e ($E_{\min} \leq e \leq E_{\max}$ is the *exponent*). The difference introduced by changing the last binary digit of the mantissa is $\pm s.\varepsilon_{\text{last}}$ where $\varepsilon_{\text{last}} = 2^{-(p-1)}$: the *unit in the last place* or *ulp*. Such a decomposition is unique for a given number if we impose that the leftmost digit of the mantissa is 1 — this is called a *normalised representation*. Except in the case of numbers of very small magnitude, IEEE-754 always works with normalised representations. There exists a least positive normalised number m_{normal} and a least positive denormalized number m_{denormal}, and the denormals are the multiples of m_{denormal} less than m_{normal}. All representable numbers are multiples of m_{denormal}.

We shall now attempt to define an imprecise axiomatisation of the relationship between the result of a floating-point operation and the result of the corresponding ideal operation over real numbers. For this, we shall distinguish operations plus and minus on the one hand, multiplication and division on the other hand, since the former have properties of which we can take advantage.

Let us consider addition or subtraction $x = \pm a \pm b$. Suppose that $0 \leq x \leq m_{\text{normal}}$. a and b are multiples of m_{denormal} and thus $a - b$ is exactly represented as a denormalized number; therefore $r(x) = x$. If $x > m_{\text{normal}}$, then $|r(x) - x| \leq \varepsilon_{\text{rel}}.x$. In other words, if the result of an addition or subtraction is denormal, then it is exact.[12] The cases for $x \leq 0$ are symmetrical.

We therefore obtain the following axiomatisation of the rounding of a positive real number x, result of a floating-point addition or subtraction, to a floating-point value r, using round-to-nearest:

$$\mathit{Round}^{\pm}_{+}(r, x) \triangleq (x \leq m_{\text{normal}} \wedge r = x) \vee (x > m_{\text{normal}} \wedge -\varepsilon_{\text{rel}}.x \leq r - x \leq \varepsilon_{\text{rel}}.x) \tag{4.1}$$

[11]To be pedantic, since IEEE floating-point formats are of a finite size, the rounding operation could be exactly represented by enumeration of all possible cases; this would anyway be impossible in practice due to the enormous size of such an enumeration.

[12]William Kahan has written extensively about the advantages of the existence and proper handling of denormals, otherwise known as *gradual underflow*, as opposed to flushing to zero all numbers too small to be approximated by normal numbers, a process known as *flush to zero*. For instance, with gradual underflow, $a \ominus b = 0$ is equivalent to $a = b$, quite a desirable property. See e.g. [63, p. 6]. Unfortunately, certain architectures do not implement gradual underflow, for the sake of efficient; then one has to use the *Round* and not the *Round*$^{\pm}$ predicate.

We then use this predicate to construct an axiomatisation for rounding of numbers of any sign:

$$\begin{aligned}\mathit{Round}^{\pm}(r,x) \triangleq (x=0 \wedge r=0) \vee (x>0 \wedge \mathit{Round}^{\pm}_{+}(r,x)) \vee \\ (x<0 \wedge \mathit{Round}^{\pm}_{+}(-r,-x)) \quad (4.2)\end{aligned}$$

Equivalently, this last formula can be simplified into the equivalent:

$$\begin{aligned}\mathit{Round}^{\pm}(r,x) \triangleq (-m_{\text{normal}} \leq x \leq m_{\text{normal}} \wedge x=r) \\ \vee (x > m_{\text{normal}} \wedge x(1-\varepsilon_{\text{rel}}) \leq r \leq x(1+\varepsilon_{\text{rel}})) \\ \vee (x < -m_{\text{normal}} \wedge x(1+\varepsilon_{\text{rel}}) \leq r \leq x(1-\varepsilon_{\text{rel}})) \quad (4.3)\end{aligned}$$

Consider now multiplication or division $x = a \otimes b$ or $x = a \oslash b$. Here, we cannot assume that if the result is denormal, then it is exact. A correct axiomatization of rounding for positive numbers is:

$$\begin{aligned}\mathit{Round}_{+}(r,x) \triangleq (x \leq m_{\text{normal}} \wedge r \geq 0 \wedge x-\varepsilon_{\text{abs}} \leq r \leq x+\varepsilon_{\text{abs}}) \\ \vee (x > m_{\text{normal}} \wedge -\varepsilon_{\text{rel}}.x \leq r-x \leq \varepsilon_{\text{rel}}.x) \quad (4.4)\end{aligned}$$

where $\varepsilon_{\text{abs}} = m_{\text{denormal}}/2$. Now for rounding for any sign:

$$\begin{aligned}\mathit{Round}(r,x) \triangleq (x=0 \wedge r=0) \vee (x>0 \wedge \mathit{Round}_{+}(r,x)) \vee \\ (x<0 \wedge \mathit{Round}_{+}(-r,-x)) \quad (4.5)\end{aligned}$$

To each floating-point expression e, we associate a "rounded-off" variable r_e, the value of which we constrain using $\mathit{Round}^{\pm}(r_e,e)$ or $\mathit{Round}(r_e,e)$. For instance, a expression $e = a \oplus b$ is replaced by a variable r_e, and the constraint $\mathit{Round}^{\pm}(r_e, a+b)$ is added to the semantics. In the case of a compound expression $e = ab + c$, we introduce $e_1 = ab$, and we obtain $\mathit{Round}^{\pm}(r_e, r_{e_1}+c) \wedge \mathit{Round}(r_{e_1}, ab)$. If we know that the compiler uses a fused multiply-add operator, we can use $\mathit{Round}(r_e, ab+c)$ instead.

The drawbacks of such axiomatization of floating-point operations are that they introduce case disjunctions into formulas, leading to extra work for quantifier elimination procedures, and also that they make very unnatural coefficients appear — that is, rational numbers with large numerator and denominator, such as $1+\varepsilon_{\text{rel}}$ or $1-\varepsilon_{\text{rel}}$, or very large integers such as $1/m_{\text{normal}}$. To go back to our very simple rate limiter running example (§3.4.3), analysis times are multiplied by 12 if one stops assuming that floating-point variables behave like reals, and instead uses some axiomatization [81, p. 9]. Furthermore, the formula produced is very large and hardly readable, doing many case disjunctions. Perhaps it is possible to simplify such large formulas by allowing some limited overapproximation — for instance, we can replace the function mapping p to p' as defined by $(0 \leq p \leq 1 \leq p' = p) \vee (p > 1 \leq p' = (1+\epsilon)p)$ by the function defined by $0 \leq p \wedge p' = (1+\epsilon)p$, since the latter formula always gives a solution for p' greater or equal to that of the former. Even better, such simplifications could be performed during the elimination procedure. Further investigations are needed in that respect.

4.6. **Integers.** We have mentioned in §2.2.2 that Presburger arithmetic admits quantifier elimination. We therefore could apply quantifier elimination in Presburger arithmetic, similarly as we do with respect to real linear arithmetic. Unfortunately, as we shall see in §7.1, such an approach suffers from explosion in the size of the formulas and the cost of the algorithms.

Instead, we used a *relaxation* approach: all integers are treated as reals; strict inequalities $a < b$ where both sides are integers are recoded $a \leq b - 1$. For instance, if the program contains a if-then-else over $x \leq y$, then $x \leq y$ is a precondition for the "then" branch, and $x \geq y + 1$ is a precondition of the "else" branch. Note that this means that traces of execution such that $y < x < y + 1$ are considered to fail.

In some cases, such as the McCarthy 91 function example from §5.2, it is necessary to constraint the reasoning procedures so that they consider that the negation of $a \leq b$ is $a \geq b + 1$. We hope that improvements of quantifier elimination algorithms will be able to allow a more elegant approach.

Another issue is that in many programming languages, integers are bounded and that arithmetic operations are actually performed modulo 2^n (with n typically 8, 16, 32 or 64). The problem then lies within an enormous, but finite, state space. Clever techniques for reasoning about *bit-vector arithmetic* are being investigated by the SMT-solving community. Again, we hope that future work will provide good quantifier elimination techniques for this arithmetic, or combinations thereof with the linear theory of reals.

4.7. **Nonlinear constructs.** In §7.2 we explain how to fully deal with nonlinear program constructs and templates, at the expense of very high computational complexity.

A more practical approach is to linearise the program expressions [78][76, ch. 6]. If we encounter an assignment $z := x * y$ and we know, perhaps through rough interval analysis, an interval $y \in [y_{\min}, y_{\max}]$, we can use the following abstraction of the semantics of this assignment:

$$(x \geq 0 \wedge xy_{\min} \leq z' \leq xy_{\max}) \vee (x < 0 \wedge xy_{\max} \leq z' \leq xy_{\min}) \tag{4.6}$$

If we know intervals for both x and y, then we can apply Eq. 4.6 to both $(x, [y_{\min}, y_{\max}])$ and $(y, [x_{\min}, x_{\max}])$, and take the conjunction of the resulting formulas.

5. Complex control flow

We have so far assumed no procedure call, and at most one single loop. We shall see here how to deal with arbitrary control flow graphs and call graph structures.

5.1. **Arbitrary control graph and loop nests.** In Sec. 3.4, we have explained how to abstract a single fixed point. The method can be applied to multiple nested fixed points by replacing the inner fixed point by its abstraction. For instance, assume the rate limiter of Sec. 3.4.3 is placed inside a larger loop. One may replace it by its abstraction:

```
if (e1max > e3max) {
  s1max = e1max;
} else {
  s1max = e3max;
}
assume(s1 <= s1max);
```

Listing 10: Loop nest

```
i=0;
while(true) { /* A */
  if (choice()) {
    j=0;
    while(j < i) { /* B */
      /* something */
      j=j+1;
    }
    i=i+1;
    if (i==20) {
      i=0;
    }
  } else {
    /* something */
  }
}
```

```
/* and similar for s1min */
```

Alternatively, we can extend our framework to an arbitrary control flow graph with nested loops, the semantics of which is expressed as a single fixed point. We may use the same method as proposed by Gulwani et al. [56, §2] and other authors. First, a *cut set* of program locations is identified; any cycle in the control flow graph must go through at least one program point in the cut set. In widening-based fixed point approximations, one classically applies widening at each point in the cut set. A simple method for choosing a cut set is to include all targets of back edges in a depth-first traversal of the control-flow graph, starting from the start node; in the case of structured program, this amounts to choosing the head node of each loop. This is not necessarily the best choice with respect to precision, though [56, §2.3]; Bourdoncle [18, Sec. 3.6] discusses methods for choosing such as cut-set.

To each point in the cut set we associate an element in the abstract domain, parameterised by a number of variables. The values of these variables for all points in the cut-set define an invariant candidate. Since paths between elements of the cut sets cannot contain a cycle, their denotational semantics can be expressed simply by an existentially quantified formula. Possible paths between each source and destination elements in the cut-set defined a stability condition (Formula 3.6). The conjunction of all these stability conditions defines acceptable inductive invariants. As above, the least inductive invariant is obtained by writing a minimisation formula (Sec. 3.4).

Let us consider the loop nest in Listing 10. We choose program points A and B as cut-set. At program point A, we look for an invariant of the form $I_A(i,j) \triangleq i_{\min,A} \leq i \leq i_{\max,A}$, and at program point B, for an invariant of the form $I_B(i,j) \triangleq i_{\min,B} \leq i \leq i_{\max,B} \wedge j_{\min} \leq j \leq j_{\max} \wedge \delta_{\min} \leq i - j \leq \delta_{\max}$ (a *difference-bound* invariant). The (somewhat edited for brevity) stability formula is written:

$$\begin{aligned}\forall j\ I_A(0,j) \wedge \forall i \forall j\ ((I_B(i,j) \wedge j \geq i \wedge (i+1 \leq 19 \vee \\ i+1 = 20 \vee i+1 \geq 21)) \Rightarrow \text{If}[i+1=20, I_A(0,j), I_A(i+1,j)]) \wedge \\ \forall i \forall j\ (I_A(i,j) \Rightarrow I_B(i,0)) \wedge \forall i \forall j\ ((I_B(i,j) \wedge j < i) \\ \Rightarrow I_B(i,j+1)) \quad (5.1)\end{aligned}$$

where $\text{If}[b, e_1, e_2] = (b \wedge e_1) \vee (\neg b \wedge e_2)$.

Replacing I_A and I_B into this formula, then applying quantifier elimination, we obtain a formula defining all acceptable tuples $(i_{\min,A}, i_{\max,A}, i_{\min,B}, i_{\max,B}, j_{\min}, j_{\max}, \delta_{\min}, \delta_{\max})$. Optimal values are then obtained by further quantifier elimination: $i_{\min,A} = i_{\min,B} = j_{\min} = 0$, $i_{\max,A} = i_{\max,B} = 19$, $j_{\max} = 20$, $\delta_{\min} = 1$, $\delta_{\max} = 19$.

The same example can be solved by replacing 20 by another variable `n` as in Sec. 3.4.2.

5.2. **Procedures and Recursive Procedures.**

We have so far considered abstractions of program blocks with respect to sets of program states. A program block is considered as a transformer from a state of input program states to the corresponding set of output program states. The analysis outputs a sound and optimal (in a certain way) abstract transformer, mapping an abstract set of input states to an abstract set of output states.

Assuming there are no recursive procedures, procedure calls can be easily dealt with. We can simply inline the procedure at the point of call, as done in e.g. ASTRÉE [14, 15, 37]. Because inlining the concrete procedure may lead to code blowup, we may also inline its abstraction, considered as a nondeterministic program. Consider a complex procedure `P` with input variable `x` and output variable `x`. We abstract the procedure automatically with respect to the interval domain for the postcondition ($z_{\min} \leq z \leq z_{\max}$); suppose we obtain $z_{\max} := 1000; z_{\min} := x$ then we can replace the function call by `z <= 1000 && z >= x`. This is a form of *modular interprocedural analysis*: considering the call graph, we can abstract the leaf procedures, then those calling the leaf procedures and so on. We can also do likewise for nested loops: abstract the innermost loop, then the next innermost one, etc. This method is however insufficient for dealing with recursive procedures.

In order to analyse recursive procedures, we need to abstract not sets of states, but sets of pairs of states, expressing the input-output relationships of procedures. In the case of recursive procedures, these relationships are the least solution of a system of equations.

To take a concrete example, let us consider McCarthy's famous "91 function" [70, 71], which, non-obviously, returns 91 for all inputs less than 101:

```
int M(int n) {
  if (n > 100) {
    return n-10;
  } else {
    return M(M(n+11));
  }
}
```

The concrete semantics of that function is a relationship R between its input n and its output r. It is the least solution of

$$R \supseteq \{(n,r) \in \mathbb{Z}^2 \mid (n > 100 \wedge r = n - 10) \vee \\ (n \leq 100 \wedge \exists n_2 \in \mathbb{Z} (n+11, n_2) \in R \wedge (n_2, r) \in R)\} \quad (5.2)$$

We look for a inductive invariant of the form $I \triangleq ((n \geq A) \wedge (r - n \geq \delta) \wedge (r - n \leq \Delta)) \vee ((n \leq B) \wedge (r = C))$, a non-convex domain (Sec. 4.3). By replacing R by I into inclusion 5.2, and by universal quantification over n, r, n_2, we obtain the set of admissible parameters for invariants of this shape. By quantifier elimination, we obtain $(C = 91) \wedge (\delta = \Delta = -10) \wedge (A = 101) \wedge (B = 100)$ within a fraction of a second using Mjollnir (see Sec. 6).

In this case, there is a single acceptable inductive invariant of the suggested shape. In general, there may be parameters to optimise, as explained in Sec. 3.4. The result of this analysis is therefore a map from parameters defining sets of states to parameters defining sets of pairs of states (the abstraction of a transition relation). This abstract transition relation (a subset of $X \times Y$ where X and Y are the input and output state sets) can be transformed into an abstract transformer in $X^\sharp \to Y^\sharp$ as explained in Sec. 3.2. Such an interprocedural analysis may also be used to enhance the analysis of loops [72].

6. Implementations and Experiments

We have implemented the techniques of Sec. 3 in quantifier elimination packages, including Mathematica[13] and Reduce 3.8[14] + Redlog[15] in addition to our own package, Mjollnir [81].[16] We ignore which exact techniques are implemented within Mathematica. [17] Redlog appears to implement some virtual substitution method [44, 107].

As test cases, we took a library of operators for synchronous programming, having streams of floating-point values as input and outputs. These operators are written in a restricted subset of C and take as much as 20 lines. A front-end based on CIL [85] converts them into formulas, then these formulas are processed and the corresponding abstract transfer functions are pretty-printed. Since for our application, it is important to bound numerical quantities, we chose the interval domain.

Among the extensions in §4, we implemented those relevant to floating-point (§4.5) and integers (§4.6), and did more manual experiments with infinities (§4.2) and recursive functions (§5.2).

For instance, the rate limiter presented in Sec. 3.4.3 was extracted from that library. Since this operator includes a memory (a variable whose value is retained from a call to the operator to the next one), its data-flow semantics is expressed using a fixed-point. When considered with real variables, the resulting expanded formula was approximately

[13]Mathematica is a commercial computer algebra package available under an unfree license from Wolfram Research [108].

[14]Reduce is a computer algebra package from Anthony C. Hearn, now available under a modified BSD licence.

[15]Redlog is an extension to Reduce for working over quantified formulas.

[16]Mjollnir is available under a free license from the author's home page. In addition to the author's own quantifier elimination techniques, it implements Ferrante and Rackoff and Loos and Weispfenning's.

[17]Loos and Weispfenning's quantifier elimination procedure is used by Mathematica to perform simplifications over linear inequalities [108, §A.9.5], but we are unsure whether this is the algorithm called by the `Reduce` function.

1000 characters long, and with floating point variables approximately 8000 characters long. Despite the length of these formulas, they can be processed by Mjollnir in a matter of seconds. The result can then be saved once and for all.

Analysers such as Astrée [14, 15, 37] must have special knowledge about such operators, otherwise the analysis results are too coarse (for instance, the intervals do not get stabilized at all). The Astrée development team therefore had to provide specialized, hand-written analyses for certain operators. In contrast, all linear floating-point operators in the library were analysed within a fraction of a second using the method in the present article, assuming that floating-point values in the source code were real numbers. If one considered instead the abstraction of floating-point computations using real numbers from Sec. 4.5, computation times did not exceed 17 seconds per operator; the formulas produced are considerably more complex than in the real case. Note that this computation is done once and for all for each operator; a static analyser can therefore cache this information for further use and need not recompute abstractions for library functions or operators unless these functions are updated.

Our analyser front-end currently cannot deal with non-numerical operations and data structures (pointers, records, and arrays). It is therefore not yet capable of directly dealing with the real control programs that e.g. Astrée accepts, which do not consist purely of numerical operators. We plan to integrate our analysis method into a more generic analyser. Alternatively, we plan to adapt a front-end for synchronous programming languages such as Simulink, a tool widely used by control/command engineers.

The correctness of the methods described in this article does not rely on any particularity of the quantifier elimination procedure used, provided one also has symbolic computation procedures for e.g. putting formulas in disjunctive normal form and simplifying them. The difference between the various quantifier elimination and simplification procedures is efficiency; experiments showed that ours was vastly more efficient than the others tested for this kind of application. For instance, our implementation of our quantifier elimination algorithm [81] was able to complete the analysis of the rate limiter of Sec. 3.4.3, implemented over the reals, in 1.4 s, and in 17 s with the same example over floating-point numbers, while Redlog took 182 s for the former and could not finish the latter, and Mathematica could analyse neither (out-of-memory). On other examples, our quantifier elimination procedure is faster than the other ones, or can complete eliminations that the others cannot.

7. Extensions to other numerical domains

We have so far concerned ourselves solely with real (and possibly Boolean) variables appearing in linear arithmetic formulas. In §4, we have seen how to reason over certain other data types (integers, floating-point values), but again modeling them as real numbers. Yet, in §2.2, we pointed out that linear real arithmetic is not the only arithmetic theory with quantifier elimination algorithms; other well-known examples include Presburger arithmetic (linear integer arithmetic) and the theory of real closed fields (that is, polynomial real arithmetic).

In this section, we report on using quantifier elimination in both these theories for computing optimal transformers or fixed points. Unfortunately, experiments have shown that the high complexity of the quantifier elimination algorithms for these theories and the lack of simplifications for the formulas they produce preclude their use in practice. The results in this section are thus mainly of theoretical interest.

7.1. **Presburger arithmetic.** The approach from §4.6 relaxes integers to reals. It therefore yields correct results, but might lead to overapproximations that could be avoided if integers were used instead. What if we used quantifier elimination on Presburger arithmetic instead?

Consider the following simple example:

```
int a, m;
...
i = 0;
while (i < m) {
  i = i+a;
}
```

Let us abstract the loop variable a using an interval $[l, h]$. These bounds must satisfy $I \triangleq l \leq 0 \leq h$ (initialisation of the loop) and

$$S \triangleq \forall i \ \ (l \leq i \leq u \Rightarrow (i > m \vee l \leq i + a \leq u)) \tag{7.1}$$

The condition for the existence of a finite range of values for i is therefore $\exists l \exists u \ I \wedge S$.[18] Intuitively, this condition is equivalent to $m < 0 \vee a \geq 0$. Yet, the formula produced by the quantifier elimination for Presburger arithmetic from Redlog yields a very large formula, more than one page long, which we have therefore omitted.[19] Redlog cannot even perform simplifications such as replacing $i = 0 \vee i = 1 \vee i = 2$ by $0 \leq i \leq 2$.

In comparison, the real relaxation gives exact and fast results:

$$S_{\mathbb{R}} \triangleq \forall i \ \ (i < l \vee i > u \vee i \geq m + 1 \vee l \leq i + a \leq u)) \tag{7.2}$$

Eliminating quantifiers from $\exists l \exists u \ I \wedge S_{\mathbb{R}}$ yields immediately the answer $a \geq 0 \vee m \leq -1$.

7.2. **Real polynomial constraint domains.** We now consider the abstraction of program states (in $\mathbb{R}^V$) using domains defined by polynomial constraints, a natural extension of those seen previously (§3.1); the orthogonal extensions from § 4 also apply. Instead of quantifier elimination in linear real arithmetic, we shall use quantifier elimination in the theory of real closed fields. One difference, though, is that we will not be able to produce nice, closed form formulas, at least not in general.

7.2.1. *Method.* We generalise the constructs of §3, except those of § 3.3, to formulas over polynomial inequalities. The same results hold:

- For any loop-free program code, and any template polynomial abstract domain with parameters $p_1, \dots, p_n$, there is a family of formulas $F_1, \dots, F_n$ that uniquely defines the optimal parameters $p'_1, \dots, p'_{n'}$ of the postcondition with respect those $p_1, \dots, p_n$ in the precondition (the free variables of F_i are among $p_1, \dots, p_n, p'_i$).

[18]This example is motivated by the fact that for $a \neq 0$, the loop terminates if and only if $[l, u]$ is finite. It is a simplified version of a loop termination problem from Paul Feautrier and Laure Gonnord.

[19]It could be that Redlog gives erroneous answers. At some point, we generated a formula F with free variables a, m such that Redlog produced falsewhen eliminating quantifiers from $\exists m \exists a \ F$, and produced truewhen eliminating quantifiers from $\exists a \exists m \ F$; at another point we made Reduce/Redlog crash with a segmentation fault.

- For any loop, and any template polynomial abstract domain with parameters p_1, $\dots, p_n$, there is a family of formulas $F_1, \dots, F_n$ that uniquely defines the optimal parameters $p'_1, \dots, p'_{n'}$ of the least inductive invariant for that loop, with respect those $p_1, \dots, p_n$ in the precondition (the free variables of F_i are among $p_1, \dots, p_n, p'_i$).

The main obstacle is the high cost of quantifier elimination in the theory of real closed fields. The other crucial difference is that it is in general impossible to move from such a formula to a formula computing p'_i from $p_1, \dots, p_n$, as we did in § 3.3. By performing the cylindrical algebraic decomposition with variables $p_1, \dots, p_n$ first, we could obtain the tree structure with case disjunctions, as the output of Algorithm 1. But at the leaves, we would obtain formulas defining p'_i as a specific root of a polynomial in the variable p'_i, with coefficients themselves polynomials in $p_1, \dots, p_n$.

In Sec. 3.3, we explained how to turn a formula over linear arithmetic defining a partial function from $p_1, \dots, p_n$ to p'_i — that is, a relation between $(p_1, \dots, p_n, p'_i)$ such that for any choice of $p_1, \dots, p_n$ there is at most one suitable p'_i — into an algorithmic function, expressed using linear assignments and if-then-else over linear inequalities. Can we do the same here? That is, can we turn a formula over nonlinear arithmetic defining a partial function from $p_1, \dots, p_n$ to p'_i into a simple algorithm written using, say, if-then-else tests and normal arithmetic operators as well as the n-th root operations $\sqrt[n]{}$? For instance, if we have a formula $p' \geq 0 \wedge p'^2 = 2p$, we would like to obtain $p' = \sqrt{2p}$.

Unfortunately, this is impossible in the general case. The Abel-Ruffini theorem, from Galois theory, states that for polynomials in one variable of degrees higher or equal to 5, there is in general no way to express the value of the roots using only arithmetic operations ($+$, $-$, $\times$, $/$) and radicals ($\sqrt[n]{}$). Thus, we cannot hope to obtain in general a simple algorithm expressing p'_i as a function of $p_1, \dots, p_n$ using tests, arithmetic operations ($+$, $-$, $\times$, $/$) and radicals ($\sqrt[n]{}$).

Let us now assume that there are no precondition parameters $p_1, \dots, p_n$ or, equivalently, that we know exactly their value. Would it be at least possible to compute the values of the p'_i?

p'_i is defined as the only solution of a logical formula with a single free variable, built using polynomial arithmetic. By putting this formula into DNF, we can reduce the problem to computing the only solution, if any, of a conjunction of polynomial inequalities and equalities. Such a solution can be computed to arbitrary precision; in fact, for any ϵ, one can obtain bounds $[l, h]$ such that $l \leq p'_i \leq h$ and $h - l \leq \epsilon$. Unfortunately, the cost of such computations is high. [10]

7.2.2. *Experiments.* Consider $l_x \leq x \leq h_x$, $l_y \leq y \leq h_y$ and the problem of generating the optimal abstract transfer function for the multiplication operation, $z := x * y$. We wish to obtain l_z and h_z such that $l_z \leq z \leq h_z$, and l_z and h_z are optimal (l_z is maximal, h_z is minimal). We first define the set of admissible (not necessarily minimal) h_z:

$$A \triangleq \forall x \forall y (l_x \leq x \leq h_x \wedge l_y \leq y \leq h_y \Rightarrow xy \leq h_z) \tag{7.3}$$

Now we define the *least* value for h_z:

$$O \triangleq A \wedge \forall h \; (A[h/h_z] \Rightarrow h \geq h_z) \tag{7.4}$$

The free variables of this formula are l_x, h_x, l_y, h_y and h_z.

Mathematica 7 performs quantified elimination by cylindrical algebraic decomposition on this formula in 4.3 s and yields a large formula (Fig. 7) with many case disjunctions,

$$
\begin{gathered}
\bigg(l_y < 0 \wedge \bigg(\bigg(h_y = l_y \wedge h_x \geq \frac{l_x l_y}{h_y} \wedge h_z = l_x\ l_y\bigg) \\
\vee(l_y < h_y \leq 0 \wedge ((l_x \leq 0 \wedge\ h_x \geq l_x \wedge h_z = l_x l_y) \vee\ (l_x > 0 \wedge h_x \geq l_x \wedge h_z = h_y\ l_x))) \\
\vee \bigg(h_y > 0 \wedge \bigg(\bigg(l_x < 0 \wedge\ \bigg(\bigg(l_x \leq h_x \leq \frac{l_x\ l_y}{h_y} \wedge h_z = l_x l_y\bigg) \vee\ \bigg(h_x > \frac{l_x l_y}{h_y} \wedge\ h_z = h_x h_y\bigg)\bigg)\bigg) \\
\vee(l_x = 0 \wedge\ h_x \geq 0 \wedge h_z = h_x h_y) \vee (l_x > 0 \wedge\ ((h_x = l_x \wedge h_z = h_y l_x) \\
\vee(h_x > l_x \wedge h_z = h_x\ h_y))))))) \\
\vee (l_y = 0 \wedge\ ((h_y = 0 \wedge h_x \geq l_x \wedge h_z = 0) \vee\ (h_y > 0 \wedge ((l_x < 0 \wedge ((l_x \leq h_x \leq 0 \wedge\ h_z = 0) \\
\vee (h_x > 0 \wedge h_z = h_x h_y))) \vee (l_x = 0 \wedge h_x \geq 0 \wedge h_z = h_x h_y) \vee\ (l_x > 0 \wedge ((h_x = l_x \wedge h_z = h_y l_x) \\
\vee (h_x > l_x \wedge h_z = h_x\ h_y))))))) \\
\vee \bigg(l_y > 0 \wedge\ \bigg(\bigg(h_y = l_y \wedge \bigg(\bigg(l_x < 0 \wedge\ \bigg(\bigg(h_x = \frac{l_x l_y}{h_y} \wedge\ h_z = l_x l_y\bigg) \\
\vee \bigg(h_x > \frac{l_x\ l_y}{h_y} \wedge h_z = h_x\ h_y\bigg)\bigg)\bigg) \vee (l_x = 0 \wedge h_x \geq 0 \wedge\ h_z = h_x h_y) \vee \\
\bigg(l_x > 0 \wedge\ \bigg(\bigg(h_x = \frac{l_x l_y}{h_y} \wedge\ h_z = l_x l_y\bigg) \vee \bigg(h_x > \frac{l_x\ l_y}{h_y} \wedge h_z = h_x\ h_y\bigg)\bigg)\bigg)\bigg)\bigg) \\
\vee(h_y > l_y \wedge\ ((l_x < 0 \wedge ((h_x = l_x \wedge h_z = l_x\ l_y) \vee (l_x < h_x \leq 0 \wedge h_z = h_x\ l_y) \\
\vee(h_x > 0 \wedge h_z = h_x h_y))) \vee\ (l_x = 0 \wedge h_x \geq 0 \wedge h_z = h_x h_y) \\
\vee(l_x > 0 \wedge ((h_x = l_x \wedge h_z = h_y\ l_x) \vee (h_x > l_x \wedge h_z = h_x\ h_y))))))) \,. \qquad (7.5)
\end{gathered}
$$

Figure 7: This formula is the result of quantifier elimination. It defines h_z to be the least upper bound of xy for $x \in [l_x, h_x]$ and $y \in [l_y, h_y]$.

most notably on the sign of l_x, h_x, l_y, h_y. This is quite natural: the monotonicity of the function $y \mapsto xy$ changes according to the sign of x. This function is equivalent to the much more terse

$$h_z = \max(l_x l_y, h_x l_y, l_x h_y, h_x h_y) \qquad (7.6)$$

This illustrates the limit of our approach on nonlinear problems: even on simple program constructions and with simple invariants, quantifier elimination takes nonneglible time and outputs complicated formulas. We therefore did not pursue this direction further.

8. Related work

Since the first numerical abstract analysis techniques were proposed in the 1970s, there has been considerable work on improving precision, efficiency, or both. Without attempting to be exhaustive, we shall now describe a few of the approaches and how they differ from ours.

8.1. **Relational abstract domains and modular analysis.** There is a sizeable amount of literature concerning relational numerical abstract domains; that is, domains that express constraints between numerical variables. Convex polyhedra were proposed in the 1970s [36, 58], and there have been since then many improvements to the technique; a bibliography was gathered by Bagnara et al. [4]. Algorithms on polyhedra are costly and thus a variety of domains intermediate between simple interval analysis and convex polyhedra were proposed [25, 74, 99].

It is possible to use relational abstract domains such as polyhedra to model the input/output relationship of a program, function or block [58, §7.2, p. 112]. Instead of considering only the current values of the program variables $(v_1, \ldots, v_n)$ at the various program points, one also considers the initial values $(v_1^0, \ldots, v_n^0)$ of these variables at the beginning of the program, function or block; thus the computed polyhedra, for instance, relate $(v_1^0, \ldots, v_n^0, v_1, \ldots, v_n)$. We employed this approach when dealing with recursive procedures (§5.2). Such an approach is modular: one can for instance analyse a procedure in such a way, and plug the result of the analysis at each point of call; though of course one loses optimality. It also provides for modular analysis of loop nests: one first analyses the innermost loop, and then replace this innermost loop by the result of the analysis, considered as a nondeterministic program; one then proceeds to the next innermost loop.

One limit of this approach is that the relationship between input and output is constrained by the abstract domain. Most numerical abstract domains concern *convex* relations: difference bound constraints, octagons, polyhedra etc. are all geometrically convex (given two points a, b in the concretisation, the segment $[a, b]$ is also in the concretisation). Note that the result of the analysis of the absolute value function (§3.2), as expressed by Rel. 3.5, or that of the rate limiter (§3.4.3), are piecewise linear but not convex.

The idea of producing procedure summaries [103] as formulas mapping input bounds to output bounds is not new. Rugina and Rinard [97], in the context of pointer analysis (with pointers considered as a base plus an integer offset), proposed a reduction to linear programming. This reduction step, while sound, introduces an imprecision that is difficult to measure in advance; our method, in contrast, is guaranteed to be “optimal” in a certain sense. Rugina and Rinard’s method, however, allows some nonlinear constructs in the program to be analysed. Martin et al. [72] proposed applying interprocedural analysis to loops.

Seidl et al. [102] also produce procedure summaries as numerical constraints. Our procedure summaries are implementations of the corresponding abstract transformer over some abstract domain, while theirs outputs a relationship between input and output concrete values. Their analysis considers a *convex* set of concrete input-output relationships, expressed as *simplices*, a restricted class of convex polyhedra. This restriction trades precision for speed: the generator and constraint representations of simplices have approximately the same size, while in general polyhedra exponential blowup can occur. Tests by arbitrary linear constraints cannot be adequately represented within this framework. Seidl et al. [102, Sec. 4] propose deferring those constraints using auxiliary variables; this, however, loses some precision. Their analysis and ours are therefore incomparable, since they make different choices between precision and efficiency.

Lal et al. [67] proposed an interprocedural analysis of numerical properties of functions using weighted pushdown automata. The “weights” are taken in a finite height abstract domain, while the domains we consider have infinite height.

8.2. **Computations of exact fixed points.** The limitations of the widening approach explained in §2.1 have been recognized for long. There has therefore been extensive research about computing precise inductive invariants, if possible the least inductive invariant inside the abstract domain considered.

Several methods have been proposed to synthesize invariants without using widening operators [29, 32, 98]. In common with us, they express as constraints the conditions under which some parametric invariant shape truly is an invariant, then they use some resolution or simplification technique over those constraints. Again, these methods are designed for solving the problem for one given set of constraints on the inputs, as opposed to finding a relation between the output or fixed-point constraints and the input constraints. In some cases, the invariant may also not be minimal.

Bagnara et al. [5, 6] proposed improvements over the "classical" widenings on linear constraint domains [58]. Gopan and Reps [54] introduced "lookahead widenings": standard widening-based analysis is applied to a sequence of syntactic restrictions of the original program, which ultimately converges to the whole program; the idea is to distinguish phases or modes of operation in order to make the widening more precise. Gonnord [52], Gonnord and Halbwachs [53], Leroux and Sutre [68] have proposed *acceleration* techniques: when the transition relation τ is of certain particular forms, it is possible to compute its transitive closure τ^+ exactly or with small imprecision. Typically, acceleration techniques have difficulties dealing with programs where the control flow is not *flat*, for instance when there are paths through a loop body that affect the iteration variables in different ways, such as the circular buffer example (Listing 6).

Adjé et al. [1], Costan et al. [31], Gaubert et al. [49] proposed a "policy iteration" or "strategy iteration" approach,[20] by downwards iterations providing successive over-approximations of the least fixed point. Their approach can fail to converge to the least fixed point, for instance with expansive semantics such as those of the "circular buffer" example (Listing 6), though for some classes of programs it converges to the least fixed point [1].

Gawlitza and Seidl [51] proposed another policy iteration approach, which is guaranteed to provide the least fixed point of the system of abstract equations. In contrast to the above method, they use upwards iterations, so each value computed is an under-approximation of the abstract least fixed point. They extended that approach to template linear constraint domains [50]. The differences with our approach are twofold:

- Their approach computes the least fixed point of a system of min/max abstract equations, derived from the source code of the program. In intuitive terms, min's correspond to conditions (and closures operators in relational domains), and max's to "merge points" in the control flow graph (end of if-then-else). This approach thus incurs the same problem of "undistinguished paths" as the example from §2.1. Even then, the policy iteration algorithm may iterate across a number of iterations exponential in the number of merge points.

  An alternative would be to consider a cut-set for the control flow graph and distinguish each path between two points in the cut-set. in other words, for a single loop, one would consider each individual control path inside the loop body. The number of such paths is exponential in the number of tests, and thus the

[20]The terminology comes from game theory. In broad terms, their consider equations with max/min operators, which are similar to the "minimax" operators appearing in the definition of the value of games in game theory. Choosing which argument of a "min" is used corresponds, in game theory terms, of choosing a *strategy* or *policy* for a player that tries to minimise the value of the game.

"max" operation in the abstract equations would also have an exponential number of arguments. Such an approach would compute the same result as ours; however, it would be unworkable without further work to get rid of the explicitly exponential number of arguments in the "max" operation.

- Their approach needs all preconditions exactly known. In contrast, we compute it as an explicit function of the precondition. In short, our invariants are *parametric* in the precondition, while theirs are not.

Gulwani et al. [56] have also proposed a method for generating linear invariants over integer variables, using a class of templates. The methods described in the present article can be applied to linear invariants over integer variables in two ways: either by abstracting them using rationals (as in examples in Sec. 3.4.2, 5.1), either by replacing quantifier elimination over rational linear arithmetic by quantifier elimination over linear integer arithmetic, also known as Presburger arithmetic (§4.6). Gulwani et al. instead chose to first consider integer variables as rationals, so as to be able to compute over rational convex polyhedra, then bound variables and constraint parameters so as to model them as finite bit vectors, finally obtaining a problem amenable to SAT solving. Program variables *are* finite bit vectors in most industrial programming languages, and parameters to useful invariants over integer variables are often small, thus their approach seems justified. We do not see, however, how their method could be applied to programs operating over real or floating-point variables, which are the main motivation for the present article.

8.3. **Limitations of template-based approaches.** Much, if not all, of the published results on computing least inductive invariants in abstract domains, or at least abstract fixed points, deal with template domains [49, 50, 56], including intervals [31, 51]. The fundamental reasons for this are:

- The domain of convex polyhedra is not closed under infinite intersection. Thus, in general, there is no best abstraction of a set of states. In general, there is no least inductive invariant inside the domain.
- These methods replace the problem of dealing with arbitrarily complex shapes such as convex polyhedra by dealing with a vector of real numbers in finite, fixed dimension. The coefficients of these vectors are then amenable to a variety of constraint solving techniques.

A common criticism of template-based approaches, is that they suppose that one knows the interesting templates beforehands — in the case of linear constraint domains, interesting directions in space. In contrast, methods based on general convex polyhedra infer these directions themselves, through the convex hull and widening operations [36, 58]. For instance, an analysis using standard polyhedra of the following program will infer that $2x = y$, while a template based approach would succeed in doing so only if a template of the form $2x - y$ has been provided:

```
x = y = 0;
while (true) {
  x = x+1;
  y = y+2;
}
```

We understand and share that criticism. In some cases there exist "natural" templates, such as intervals, or when dealing with timing or scheduling constraints, difference bounds $v_i - v_j \leq C$, but in general, finding the correct templates seems a hard problem. A suggestion is to run iterations using general convex polyhedra and look at the stable directions of the faces of these polyhedra.

We think however that criticism of template domains should be part of wider considerations on how to choose the proper abstract domain. Abstract interpretation essentially replaces the unsolvable problem of computing the least inductive invariant of the concrete problem by the solvable problem of computing an inductive invariant in an abstract domain — in some lucky cases such as the one dealt with in this article, actually obtaining the least inductive invariant *in the abstract domain*. The choice of the abstract domain is at present somewhat arbitrary, typically hardwired into the analysis tool, at best chosen by some command-line flags.

Convex polyhedra [36, 58, 59] are popular because many programming idioms naturally exhibit convexity — for instance, the set of loop indices $(i, j, k, \dots)$ of nested loops occurring in numerical analysis programs is often convex. Yet, one can easily think of programs where some interesting properties are not convex (a test $|x| \geq 1$, for instance). There are some cases where it is important not to enforce convexity and instead implement disjunctive domains, capable of representing properties such as $x \leq -1 \vee x \geq 1$; an example is *trace partitioning* [93]. Thus, a static analysis tool based on general convex polyhedra also enforces an *a priori* convex shape that might not be representative of the useful program invariants.

While convex polyhedra are not enough, they are often "too much": their complexity is too high for many applications. Indeed, even for less costly constraint domains such as octagons [74, 77], it is simply too expensive to compute constraints between all visible program variables, so some analysers choose *a priori* to consider relations only between certain variables — an approach knowing as *packing* in the Astrée tool [37, 39]. Again, the packing choice is a form of *a priori* template guess made by the analyser, using heuristics that look at the way the program is organized.

Further research is obviously needed in how to choose and adapt abstract domains so that they can represent interesting inductive properties at reasonable costs. This problem is similar to the problem of finding the correct predicates in predicate abstraction. For this, various methods for finding predicates in addition to those syntactically present in the program have been proposed, especially those based on the analysis of spurious counterexamples (*counterexample-guided abstraction refinement* or CEGAR).[21] Perhaps similar ideas could be employed for suggesting suitable additional numerical templates, finer-grained packing or disjunctions.

8.4. **Relational domains beyond polyhedra.** In earlier works, we have proposed a method for obtaining input-output relationships of digital linear filters with memories, taking into account the effects of floating-point computations [79]. This method computes an exact relationship between bounds on the input and bounds on the output, without the need for an abstract domain for expressing the local invariant; as such, for this class of problems, it is more precise than the method from this article. This technique, however,

[21] The literature on CEGAR is too vast to be cited here without unfairness. Clarke et al. [26] introduced this approach for symbolic model checking. Notable applications to software model checking include the Blast [12] and Slam [8] tools.

cannot be easily generalized to cases where the operator block contains tests and other nonlinear constructs; the semantics of nonlinear constructs must be approximated by e.g. interval analysis.

There have been several published approaches to finding nonlinear relationships between program variables. One approach obtains polynomial equalities through computations on ideals using Gröbner bases [94, 95]. This work only deals with equalities (not inequalities), uses a classical approach of computing output constraints from a set of input constraints (instead of finding relationships between the two sets of constraints), and deals with loops using a widening operator. In comparison, our approach abstracts whole program fragments, and is modular — it is possible to "plug" the result of the analysis of a procedure at the location of a procedure call, though of course this is less precise than inlining the procedure.

Since nonlinear relations are notoriously costly to compute upon, Bagnara et al. [7] have proposed using further abstraction to be able to reduce the problem to computations over convex polyhedra.

Kapur [64] also proposed to use quantifier elimination to obtain invariants: he considers program invariants with parameters, and derives constraints over those parameters from the program. Our work improves on his by noting that least invariants of the chosen shape can be obtained, not just any invariant; that the abstraction can be done modularly and compositionally (a program fragment can be analysed, and the result of its analysis can be plugged into the analysis of a larger program), or combined into a "conventional" abstract interpretation framework (by using invariants of a shape compatible with that framework), and that the resulting invariants can be "projected" to obtain numerical quantities.

9. Conclusion and future prospects

Writing static analysers by hand has long been found tedious and error-prone. One may of course prove an existing analyser correct through assisted proof techniques, which removes the possibility of soundness mistakes, at the expense of much increased tediousness. In this article, we proposed instead effective methods to synthesize abstract domains by automatic techniques. The advantages are twofold: new domains can be created much more easily, since no programming is involved; a single procedure, testable on independent examples, needs be written and possibly formally proved correct. To our knowledge, this is the first effective proposal for generating numerical abstract domains automatically, and one of the few methods for generating numerical summaries. Also, it is also the only method so far for computing summaries of *floating-point* functions.

We have shown that floating-point computations could be safely abstracted using our method. The formulas produced are however fairly complex in this case, and we suspect that further over-approximation could dramatically reduce their size. There is also nowadays significant interest in automatizing, at least partially, the tedious proofs that computer arithmetic experts do and we think that the kind of methods described in this article could help in that respect.

We have so far experimented with small examples, because the original goal of this work was the automatic, on-the-fly, synthesis of abstract transfer functions for small sequences of code that could be more precise than the usual composition of abstract of individual instructions, and less tedious for the analysis designer than the method of pattern-matching the code for "known" operators with known mathematical properties. A further goal is the precise analysis of longer sequences, including integer and Boolean computations. We have shown in Sec. 4.4 how it was possible to partition the state space and abstract each region of the state-space separately; but naive partitioning according to n Booleans leads to 2^n regions, which can be unbearably costly and is unneeded in most cases. We think that automatic refinement and partitioning techniques [62] could be developed in that respect.

The main practical application that we envision is to be able to analyse numerical operator blocks from synchronous programming languages such as SIMULINK,[22] SCICOS,[23] LUSTRE,[24] SCADE[25] or SAO,[26] which are widely used for programming control systems [3], particularly in the automative and avionic industries. In order to obtain good analysis precision, such blocks often have to be analysed as a whole instead of decomposing them into individual components and applying individual transfer functions, as in our rate limiter example. The static analysis tool ASTRÉE [15, 37, 38, 39, 43, 104] outputs few, if any, false alarms on some classes of control programs because it has specific specialized transfer functions for certain operator blocks or coding patterns. Such transfer functions had to be implemented by hand; the techniques described in the present article could have been used to implement some of them automatically and even on-the-fly.

There are two important drawbacks to our method, which make it currently only useful for very precise analysis of small parts of programs. The first is that we need to "see" the whole of the loop or function that we are analysing, the instructions of which must belong to the class of constructs that we are capable of dealing with, or at least can be abstracted by them. In contrast, iterative techniques are more tolerant: they see the program state locally, at each program point, and the numerical analysis may easily interact with other analyses, such as pointers [14, 15]. The second issue is the high cost of quantifier elimination. Despite our work on new algorithms [81], in which we are still making progress, scalability remains an issue.

Acknowledgements

We would like to thanks the anonymous referees for careful reading.

References

[1] Assalé Adjé, Stéphane Gaubert, and Éric Goubault. Computing the smallest fixed point of nonexpansive mappings arising in game theory and static analysis of programs. preprint, arXiv:0806.1160v2, 2008. URL `http://arxiv.org/abs/0806.1160`.

[22] SIMULINK is a graphical dataflow modeling tool sold as an extension to the MATLAB numerical computation package. It allows modeling a physical or electrical environment along the computerized control system. A code generator tool can then provide executable code for the control system for a variety of targets, including generic C. SIMULINK is available from The Mathworks.

[23] SCICOS is a graphical dataflow modeling tool coming with the SCILAB numerical computation package, similar in use to SIMULINK. [22] It is available from INRIA under the GNU General Public License and also has code generation capabilities.

[24] LUSTRE is a synchronous programming language, from which code can be generated for a variety of platforms [23].

[25] SCADE is a graphical synchronous programming language derived from LUSTRE. It is available from Esterel Technologies. It was used for implementing parts of the Airbus A380 fly-by-wire systems, among others. [43, 104]

[26] SAO is an earlier industrial graphical synchronous programming language, used, for implementing parts of the Airbus A340 fly-by-wire systems [20], among others.

[2] Hirokazu Anai and Volker Weispfenning. Deciding linear-trigonometric problems. In *International symposium on symbolic and algebraic computation (ISSAC)*. ACM, 2000. ISBN 1-58113-218-2. doi: 10.1145/345542.345567.

[3] Karl Johan Åström and Björn Wittenmark. *Computer-controlled systems*. Prentice-Hall, 1997. ISBN 0-13-314899-8.

[4] Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella. *The Parma Polyhedra Library, version 0.9*. URL `http://www.cs.unipr.it/ppl`.

[5] Roberto Bagnara, Patricia M. Hill, Elena Mazzi, and Enea Zaffanella. Widening operators for weakly-relational numeric abstractions. In Hankin and Siveroni [60], pages 3–18. ISBN 3-540-28584-9. doi: 10.1007/11547662_3.

[6] Roberto Bagnara, Patricia M. Hill, Elisa Ricci, and Enea Zaffanella. Precise widening operators for convex polyhedra. *Science of Computer Programming*, 58(1-2):28–56, 2005. doi: 10.1016/j.scico.2005.02.003.

[7] Roberto Bagnara, Enric Rodríguez-Carbonell, and Enea Zaffanella. Generation of basic semi-algebraic invariants using convex polyhedra. In Hankin and Siveroni [60], pages 19–34. ISBN 3-540-28584-9. doi: 10.1007/11547662_4.

[8] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of C programs. In *PLDI*, pages 203–213. ACM, 2001. ISBN 1-58113-414-2. doi: 10.1145/378795.378846.

[9] Clark Barrett, Silvio Ranise, Aaron Stump, and Cesare Tinelli. The satisfiability modulo theories library (SMT-LIB). `www.smtlib.org`, 2008.

[10] Saugata Basu, Richard Pollack, and Marie-Françoise Roy. *Algorithms in real algebraic geometry*. Algorithms and computation in mathematics. Springer, 2nd edition, 2006. ISBN 3-540-33098-4.

[11] Bernd Becker, Christian Dax, Jochen Eisinger, and Felix Klaedtke. LIRA: handling constraints of linear arithmetics over the integers and the reals. In Werner Damm and Holger Hermanns, editors, *Computer-aided verification (CAV)*, volume 4590 of *LNCS*, pages 307–310. Springer, 2007. ISBN 978-3-540-73367-6. doi: 10.1007/978-3-540-73368-3_36.

[12] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker BLAST. *Int. J. of Software Tools for Technology Transfer (STTT)*, 9 (5–6):505–525, 2007. doi: 10.1007/s10009-007-0044-z. Special section FASE '04/05.

[13] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Advances in Computers*, 58:118–149, August 2003. doi: 10.1016/S0065-2458(03)58003-2.

[14] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. In Torben Æ. Mogensen, David A. Schmidt, and I. Hal Sudborough, editors, *The Essence of Computation: Complexity, Analysis, Transformation*, number 2566 in LNCS, pages 85–108. Springer, 2002. ISBN 3-540-00326-6. doi: 10.1007/3-540-36377-7_5.

[15] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A static analyzer for large safety-critical software. In *Programming Language Design and Implementation (PLDI)*, pages 196–207. ACM, 2003. ISBN 1-58113-662-5. doi: 10.1145/781131.781153.

[16] Bernard Boigelot, Sébastien Jodogne, and Pierre Wolper. An effective decision procedure for linear arithmetic over the integers and reals. *ACM Transactions on Computational Logic (TOCL)*, 6(3):614–633, 2005. ISSN 1529-3785. doi: 10.1145/1071596.1071601.

[17] Ahmed Bouajjani, Javier Esparza, and Oded Maler. Reachability analysis of pushdown automata: Application to model-checking. In Antoni W. Mazurkiewicz and Józef Winkowski, editors, *Concurrency Theory (CONCUR)*, volume 1243 of *Lecture Notes in Computer Science*, pages 135–150. Springer, 1997. ISBN 3-540-63141-0. doi: 10.1007/3-540-63141-0_10.

[18] François Bourdoncle. *Sémantique des langages impératifs d'ordre supérieur et interprétation abstraite*. PhD thesis, École polytechnique, Palaiseau, 1992.

[19] Aaron R. Bradley and Zohar Manna. *The Calculus of Computation: Decision Procedures with Applications to Verification*. Springer, October 2007. ISBN 3-540-74112-7.

[20] Dominique Brière and Pascal Traverse. Airbus A320/A330/A340 electrical flight controls — a family of fault-tolerant systems. In *FTCS-23 (Symposium on Fault-Tolerant Computing)*, pages 616–623. IEEE, June 1993. ISBN 0-8186-3680-7. doi: 10.1109/FTCS.1993.627364.

[21] Christopher W. Brown and James H. Davenport. The complexity of quantifier elimination and cylindrical algebraic decomposition. In *ISSAC (Symposium on Symbolic and algebraic computation)*, pages 54–60. ACM, 2007. ISBN 978-1-59593-743-8. doi: 10.1145/1277548.1277557.

[22] Stephen L. Campbell, Jean-Philippe Chancelier, and Ramine Nikoukhah. *Modeling and Simulation in Scilab/Scicos*. Springer, 2006. ISBN 0-387-27802-8.

[23] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John A. Plaice. LUSTRE: a declarative language for real-time programming. In *POPL (Symposium on Principles of programming languages)*, pages 178–188. ACM, 1987. ISBN 0-89791-215-2. doi: 10.1145/41625.41641.

[24] Bob F. Caviness and Jeremy R Johnson, editors. *Quantifier elimination and cylindrical algebraic decomposition*. Springer, 1998. ISBN 3-211-82794-3.

[25] Robert Clarisó and Jordi Cortadella. The octahedron abstract domain. In Roberto Giacobazzi, editor, *Static Analysis (SAS)*, number 3148 in LNCS. Springer, 2004. ISBN 3-540-22791-1. doi: 10.1007/b99688.

[26] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM*, 50(5):752–794, 2003. ISSN 0004-5411. doi: 10.1145/876638.876643.

[27] Edmund M. Clarke, Jr, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 1999. ISBN 0-262-03270-8.

[28] George Collins. Quantifier elimination for real closed fields by cylindric algebraic decomposition. In *Automata theory and formal languages (2nd GI conference)*, LNCS, pages 134–183. Springer, 1975. ISBN 0-387-07407-4. reprinted as [24].

[29] Michael A. Colón, Sriram Sankaranarayanan, and Henny Sipma. Linear invariant generation using non-linear constraint solving. In *Computer Aided Verification (CAV)*, number 2725 in LNCS, pages 420–433. Springer, 2003. ISBN 3-540-40524-0. doi: 10.1007/b11831.

[30] D. C. Cooper. Theorem proving in arithmetic without multiplication. In Bernard Meltzer and Donald Michie, editors, *Machine Intelligence 7*, pages 91–100. Edinburgh University Press, 1972. ISBN 0-85224-234-4.

[31] Alexandru Costan, Stephane Gaubert, Éric Goubault, Matthieu Martel, and Sylvie Putot. A policy iteration algorithm for computing fixed points in static analysis of programs. In Etessami and Rajamani [46], pages 462–475. ISBN 3-540-27231-3. doi: 10.1007/11513988_46.

[32] Patrick Cousot. Proving program invariance and termination by parametric abstraction, Lagrangian relaxation and semidefinite programming. In Radhia Cousot, editor, *Verification, Model Checking and Abstract Interpretation (VMCAI)*, number 3385 in LNCS, pages 1–24. Springer, 2005. ISBN 3-540-24297-X. doi: 10.1007/b105073.

[33] Patrick Cousot. *Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique des programmes.* State doctorate thesis, Université scientifique et médicale de Grenoble and Institut National Polytechnique de Grenoble, 1978. URL `http://tel.archives-ouvertes.fr/tel-00288657/en/`.

[34] Patrick Cousot and Radhia Cousot. Static determination of dynamic properties of programs. In *Proceedings of the Second International Symposium on Programming (1976)*, pages 106–130, Paris, 1977. Dunod. ISBN 2-04-005185-6. Also known as *Actes du deuxième colloque international sur la programmation.*

[35] Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *J. of Logic and Computation*, pages 511–547, August 1992. ISSN 0955-792X. doi: 10.1093/logcom/2.4.511.

[36] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Principles of Programming Languages (POPL)*, pages 84–96. ACM, 1978. doi: 10.1145/512760.512770.

[37] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The ASTRÉE analyzer. In Shmuel "Mooly" Sagiv, editor, *Programming Languages and Systems (ESOP)*, number 3444 in LNCS, pages 21–30. Springer, 2005. ISBN 3-540-25435-8. doi: 10.1007/b107380.

[38] Patrick Cousot, Radhia Cousot, Jerome Feret, Antoine Miné, Laurent Mauborgne, David Monniaux, and Xavier Rival. Varieties of static analyzers: A comparison with ASTRÉE. In *Theoretical Aspects of Software Engineering (TASE)*. IEEE, 2007. ISBN 0-7695-2856-2.

[39] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. Combination of abstractions in the astrée static analyzer. In Mitsu Okada and Ichiro Satoh, editors, *Advances in Computer Science — ASIAN 2006*, volume 4435 of *LNCS*, pages 272–300. Springer, 2008. ISBN 978-3-540-77504-1. doi: 10.1007/978-3-540-77505-8_23. URL `http://www.di.ens.fr/~cousot/COUSOTpapers/ASIAN06.shtml`.

[40] James H. Davenport and Joos Heintz. Real quantifier elimination is doubly exponential. *Journal of Symbolic Computation*, 5(1–2):29–35, April 1988. doi: 10.1016/S0747-7171(88)80004-X.

[41] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *TACAS*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008. ISBN 3-540-78799-2. doi: 10.1007/978-3-540-78800-3_24.

[42] Rocco de Nicola, editor. *Programming Languages and Systems (ESOP)*, volume 4421 of *LNCS*, 2007. Springer. ISBN 978-3-540-71316-6.

[43] David Delmas and Jean Souyris. Astrée: From research to industry. In Nielson and Filé [86], pages 437–451. ISBN 978-3-540-74060-5. doi: 10.1007/978-3-540-74061-2_27.

[44] Andreas Dolzmann, Andreas Seidl, and Thomas Sturm. *Redlog User Manual*, 3.1 edition, 2006. for Redlog version 3.06 and Reduce version 3.8.

[45] Bruno Dutertre and Leonardo de Moura. A fast linear-arithmetic solver for DPLL(T). In Thomas Ball and Robert B. Jones, editors, *Computer-aided verification*

(CAV), volume 4144 of *LNCS*, pages 81–94. Springer, 2006. ISBN 3-540-37406-X. doi: 10.1007/11817963_11.

[46] Kousha Etessami and Sriram K. Rajamani, editors. *Computer Aided Verification (CAV)*, number 4590 in LNCS, 2005. Springer. ISBN 3-540-27231-3. doi: 10.1007/b138445.

[47] Jeanne Ferrante and Charles Rackoff. A decision procedure for the first order theory of real addition with order. *SIAM J. on Computing*, 4(1):69–76, March 1975. ISSN 0097-5397. doi: 10.1137/0204006.

[48] Michael J. Fischer and Michael O. Rabin. Super-exponential complexity of Presburger arithmetic. In Richard Karp, editor, *Complexity of Computation*, number 7 in SIAM–AMS proceedings, pages 27–42. American Mathematical Society, 1974. ISBN 0-8218-1327-7.

[49] Stéphane Gaubert, Éric Goubault, Ankur Taly, and Sarah Zennou. Static analysis by policy iteration on relational domains. In de Nicola [42], pages 237–252. ISBN 978-3-540-71316-6.

[50] Thomas Gawlitza and Helmut Seidl. Precise relational invariants through strategy iteration. In Jacques Duparc and Thomas A. Henzinger, editors, *Computer science logic (CSL)*, volume 4646 of *LNCS*, pages 23–40. Springer, 2007. ISBN 978-3-540-74914-1. doi: 10.1007/978-3-540-74915-8_6.

[51] Thomas Gawlitza and Helmut Seidl. Precise fixpoint computation through strategy iteration. In de Nicola [42], pages 300–315. ISBN 978-3-540-71316-6. doi: 10.1007/978-3-540-71316-6_21.

[52] Laure Gonnord. *Accelération abstraite pour l'amélioration de la précision en analyse des relations linéaires*. PhD thesis, Université Joseph Fourier, October 2007. URL `http://tel.archives-ouvertes.fr/tel-00196899/en/`.

[53] Laure Gonnord and Nicolas Halbwachs. Combining widening and acceleration in linear relation analysis. In Kwangkeun Yi, editor, *Static analysis (SAS)*, volume 4134 of *LNCS*, pages 144–160. Springer, 2006. ISBN 3-540-37756-5. doi: 10.1007/11823230_10.

[54] Denis Gopan and Thomas W. Reps. Lookahead widening. In Thomas Ball and Robert B. Jones, editors, *Computer Aided Verification (CAV)*, volume 4144 of *LNCS*, pages 452–466. Springer, 2006. ISBN 3-540-37406-X. doi: 10.1007/11817963_41.

[55] Philippe Granger. Static analysis of linear congruence equalities among variables of a program. In *TAPSOFT*, volume 493 of *LNCS*. Springer, 1991. doi: 10.1007/3-540-53982-4_10.

[56] Sumit Gulwani, Saurabh Srivastava, and Ramarathnam Venkatesan. Program analysis as constraint solving. In *Programming Language Design and Implementation (PLDI)*. ACM, 2008. ISBN 978-1-59593-860-2. doi: 10.1145/1375581.1375616.

[57] Nicolas Halbwachs. Delay analysis in synchronous programs. In Costas Courcoubetis, editor, *Computer Aided Verification (CAV)*, volume 697 of *LNCS*, pages 333–346. Springer, 1993. ISBN 3-540-56922-7. doi: 10.1007/3-540-56922-7_28.

[58] Nicolas Halbwachs. *Détermination automatique de relations linéaires vérifiées par les variables d'un programme*. PhD thesis, Université scientifique et médicale de Grenoble and Institut National Polytechnique de Grenoble, 1979. URL `http://tel.archives-ouvertes.fr/tel-00288805/en/`.

[59] Nicolas Halbwachs, Yann-Erick Proy, and Pascal Raymond. Verification of linear hybrid systems by means of convex approximations. In *Static analysis (SAS)*, September 1994. ISBN 3-540-58485-4. doi: 10.1007/3-540-58485-4_43.

[60] Chris Hankin and Igor Siveroni, editors. *Static analysis (SAS)*, volume 3672 of *LNCS*, 2005. Springer. ISBN 3-540-28584-9.

[61] *IEEE standard for Binary floating-point arithmetic for microprocessor systems*. IEEE, 1985. ANSI/IEEE Std 754-1985.

[62] Bertrand Jeannet. Dynamic partitioning in linear relation analysis. application to the verification of reactive systems. *Formal Methods in System Design*, 23(1):5–37, July 2003. ISSN 0925-9856. doi: 10.1023/A:1024480913162.

[63] William Kahan. Advantages of gradual over abrupt underflow to zero. Slides of a keynote talk given at ARITH17, 2005.

[64] Deepak Kapur. Automatically generating loop invariants using quantifier elimination. In *Applications of Computer Algebra (ACA)*, 2004.

[65] Michael Karr. Affine relationships among variables of a program. *Acta Informatica*, 6(2), June 1976.

[66] Daniel Kroening and Ofer Strichman. *Decision procedures.* Springer, 2008. ISBN 978-3-540-74104-6.

[67] Akash Lal, Gogul Balakrishnan, and Thomas Reps. Extended weighted pushdown systems. In Etessami and Rajamani [46], pages 343–357. ISBN 3-540-27231-3. doi: 10.1007/11817963_32.

[68] Jérôme Leroux and Grégoire Sutre. Accelerated data-flow analysis. In *Static Analysis, 14th International Symposium, SAS 2007, Kongens Lyngby, Denmark, August 22-24, 2007, Proceedings*, volume 4634 of *Lecture Notes in Computer Science*, pages 184–199. Springer, 2007. doi: 10.1007/s10009-008-0064-3.

[69] Rüdiger Loos and Volker Weispfenning. Applying linear quantifier elimination. *The Computer Journal*, 36(5):450–462, 1993. Special issue on computational quantifier elimination.

[70] Zohar Manna and John McCarthy. Properties of programs and partial function logic. In Bernard Meltzer and Donald Michie, editors, *Machine Intelligence*, 5, pages 27–38. Edinburgh University Press, 1969. ISBN 0-85224-176-3.

[71] Zohar Manna and Amir Pnueli. Formalization of properties of functional programs. *Journal of the ACM*, 17(3):555–569, 1970. doi: 10.1145/321592.321606.

[72] Florian Martin, Martin Alt, Reinhard Wilhelm, and Christian Ferdinand. Analysis of loops. In Kai Koskimies, editor, *Compiler Construction (CC)*, volume 1383 of *LNCS*, pages 80–94. Springer, 1998. ISBN 3-540-64304-4. doi: 10.1007/BFb0026424.

[73] Yuri V. Matiyasevich. *Hilbert's Tenth Problem.* MIT Press, 1993. ISBN 0262132958.

[74] Antoine Miné. The octagon abstract domain. In Axel Simon, Andy King, and Jacob M. Howe, editors, *WCRE (Analysis, Slicing, and Transformation)*, pages 310–319. IEEE, 2001. doi: 10.1109/WCRE.2001.957836.

[75] Antoine Miné. Relational abstract domains for the detection of floating-point run-time errors. In David Schmidt, editor, *Programming Languages and Systems (ESOP)*, number 2986 in LNCS, pages 3–17. Springer, 2004. ISBN 3-540-21313-9. doi: 10.1007/b96702.

[76] Antoine Miné. *Domaines numériques abstraits faiblement relationnels.* PhD thesis, École polytechnique, 2004. URL `http://tel.archives-ouvertes.fr/tel-00136630/fr/`.

[77] Antoine Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006. doi: 10.1007/s10990-006-8609-1.

[78] Antoine Miné. Symbolic methods to enhance the precision of numerical abstract domains. In *Verification, Model Checking, and Abstract Interpretation (VMCAI'06)*, volume 3855 of *LNCS*, pages 348–363. Springer, January 2006. ISBN 3-540-31139-4. doi:

10.1007/11609773.

[79] David Monniaux. Compositional analysis of floating-point linear numerical filters. In Etessami and Rajamani [46], pages 199–212. ISBN 3-540-27231-3. doi: 10.1007/b138445.

[80] David Monniaux. Quantifier elimination by lazy model enumeration. In *Computer aided verification (CAV)*, LNCS. Springer, 2010. To appear.

[81] David Monniaux. A quantifier elimination algorithm for linear real arithmetic. In Iliano Cervesato, Helmut Veith, and Andrei Voronkov, editors, *Logic for Programming Artificial Intelligence and Reasoning (LPAR)*, volume 5330 of *LNAI*, pages 243–257. Springer, 2008. ISBN 978-3-540-89438-4. doi: 10.1007/978-3-540-89439-1_18.

[82] David Monniaux. Automatic modular abstractions for linear constraints. In Benjamin C. Pierce, editor, *Symposium on Principles of programming languages (POPL)*. ACM, 2009. ISBN 978-1-60558-379-2. doi: 10.1145/1480881.1480899.

[83] David Monniaux. Optimal abstraction on real-valued programs. In Nielson and Filé [86], pages 104–120. ISBN 978-3-540-74060-5. doi: 10.1007/978-3-540-74061-2_7.

[84] David Monniaux. The pitfalls of verifying floating-point computations. *Transactions on programming languages and systems (TOPLAS)*, 30(3):12, May 2008. ISSN 0164-0925. doi: 10.1145/1353445.1353446. URL `http://hal.archives-ouvertes.fr/hal-00128124/en/`.

[85] George C. Necula, Scott McPeak, Shree P. Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In R. Nigel Horspool, editor, *Compiler Construction (CC)*, volume 2304 of *LNCS*, pages 209–265. Springer, 2002. ISBN 3-540-43369-4. doi: 10.1007/3-540-45937-5_16.

[86] Hanne Riis Nielson and Gilberto Filé, editors. *Static analysis (SAS)*, volume 4634 of *LNCS*, 2007. Springer. ISBN 978-3-540-74060-5.

[87] Tobias Nipkow. Linear quantifier elimination. In Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *Automated reasoning (IJCAR)*, volume 5195 of *LNCS*, pages 18–33. Springer, 2008. ISBN 978-3-540-71069-1. doi: 10.1007/978-3-540-71070-7_3.

[88] Dominique Perrin. Finite automata. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, vol. B*, chapter 1. MIT Press, 1990. ISBN 0444880747.

[89] Mojżesz Presburger. Über die Vollstandigkeit eines Gewissen Systems der Arithmetik Ganzer Zahlen, in Welchem die Addition als Einzige Operation Hervortritt. In *Comptes-rendus du premier congrès des mathématiciens des pays slaves*, pages 92–101, Warsaw, 1929.

[90] William Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. In *Supercomputing '91*, pages 4–13. ACM, 1991. ISBN 0-89791-459-7. doi: 10.1145/125826.125848.

[91] Thomas W. Reps, Shmuel Sagiv, and Reinhard Wilhelm. Static program analysis via 3-valued logic. In Rajeev Alur and Doron Peled, editors, *Computer Aided Verification (CAV)*, volume 3114 of *LNCS*, pages 15–30. Springer, 2004. ISBN 3-540-22342-8. doi: 10.1007/b98490.

[92] Henry Gordon Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, March 1953.

[93] Xavier Rival and Laurent Mauborgne. The trace partitioning abstract domain. *Transactions on Programming Languages and Systems (TOPLAS)*, 29(5):26, 2007. ISSN 0164-0925. doi: 10.1145/1275497.1275501.

[94] E. Rodríguez-Carbonell and D. Kapur. Automatic generation of polynomial invariants of bounded degree using abstract interpretation. *Science of Computer Programming*, 64(1):54–75, 2007. ISSN 0167-6423. doi: 10.1016/j.scico.2006.03.003.

[95] Enric Rodríguez-Carbonell and Deepak Kapur. An abstract interpretation approach for automatic generation of polynomial invariants. In *SAS (Static Analysis)*, number 3148 in LNCS. Springer, 2004.

[96] Hartley Rogers, Jr. *Theory of Recursive Functions and Effective Computability.* MIT Press, 1987. ISBN 0-262-68052-1.

[97] Radu Rugina and Martin Rinard. Symbolic bounds analysis for pointers, array indices, and accessed memory regions. *ACM Trans. on Programming Languages and Systems (TOPLAS)*, 27(2):185–235, 2005. doi: 10.1145/349299.349325.

[98] Sriram Sankaranarayanan, Henny Sipma, and Zohar Manna. Constraint-based linear-relations analysis. In *Static Analysis (SAS)*, number 3148 in LNCS, pages 53–68. Springer, 2004. doi: 10.1007/b99688.

[99] Sriram Sankaranarayanan, Henny Sipma, and Zohar Manna. Scalable analysis of linear systems using mathematical programming. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 3385 of *LNCS*, pages 21–47. Springer, 2005. doi: 10.1007/b105073.

[100] Christoph Scholl, Stefan Disch, Florian Pigorsch, and Stefan Kupferschmid. Computing optimized representations for non-convex polyhedra by detection and removal of redundant linear constraints. In Stefan Kowalewski and Anna Philippou, editors, *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, volume 5505 of *LNCS*, pages 383–397. Springer, 2009. ISBN 978-3-642-00767-5. doi: 10.1007/978-3-642-00768-2_32.

[101] Abraham Seidenberg. A new decision method for elementary algebra. *Annals of Mathematics*, 60(2):365–374, September 1954. URL `http://www.jstor.org/stable/1969640`.

[102] Helmut Seidl, Andrea Flexeder, and Michael Petter. Interprocedurally analysing linear inequality relations. In de Nicola [42], pages 284–299. ISBN 978-3-540-71316-6. doi: 10.1007/978-3-540-71316-6_20.

[103] Micha Sharir and Amir Pnueli. Two approaches to inter-procedural data-flow analysis. In Neil Jones and Steven Muchnik, editors, *Program Flow Analysis: Theory and Application.* Prentice-Hall, 1981.

[104] Jean Souyris and David Delmas. Experimental assessment of Astrée on safety-critical avionics software. In Francesca Saglietti and Norbert Oster, editors, *SAFECOMP*, volume 4680 of *LNCS*, pages 479–490. Springer, 2007. ISBN 978-3-540-75100-7. doi: 10.1007/978-3-540-75101-4_45.

[105] Alfred Tarski. *A Decision Method for Elementary Algebra and Geometry.* University of California Press, 1951. reprinted as [106].

[106] Alfred Tarski. A decision method for elementary algebra and geometry. Technical Report 109, The RAND Corporation, 1957. Second edition (original 1948, revised 1951), reprint of [105].

[107] Volker Weispfenning. The complexity of linear problems in fields. *Journal of Symbolic Computation*, 5(1–2):3–27, February 1988. ISSN 0747-7171. doi: 10.1016/S0747-7171(88)80003-8.

[108] Stephen Wolfram. *The* Mathematica *Book.* Wolfram Research, 2005. Manual coming with version 5.2 of *Mathematica*.